TRACS Abschlussbericht

Andreas Kemnade, Arne Stahlbock, Deng Zhou, Helge Löding,
Henrik Röhrup, Marcin Dysarz, Marius Mauch, Taffou Happi,
Ruben Rothaupt, Waldemar Wockenfuß

ehemalige Teilnehmer: Feng Zheng, Jonas Jokovic, Li Qiu
Jinghong Jin, Jun Liang, Niklas Polke
Silvia Graumann, Susanne Spiller
Thomas Harlos, Wei Han, Xiaobo Chen

30.09.2005

Betreut von: Dr. Jan Bredereke
Dr. Ulrich Hannemann
Arbeitsgruppe Betriebssysteme, Verteilte Systeme

Inhaltsverzeichnis

Kapitel 1

Einleitung

Dieser Bericht dokumentiert das studentische Projekt TRACS (**TRA**in-**C**ontrol-**S**ystem) der Universität Bremen - Arbeitsgruppe Betriebssysteme und verteilte Systeme (AGBS). Um dem Leser den Einstieg in die Materie zu ermöglichen, soll in diesem Kapitel zunächst nach einer kleinen allgemeinen Vorstellung des Projektes ein Überblick über Themenstellung und Hintergründe des Projekts TRACS gegeben werden. Danach folgen die zentralen Arbeitsbereiche des Projekts TRACS mit anschließenden detaillierten Einblicken in das Projekt.

1.1 Einführung in TRACS

TRACS war von Oktober 2003 bis September 2005 ein Hauptstudiumsprojekt im Studiengang Informatik an der Universität Bremen. Ziel solcher Projekte ist es, dass Studenten über einen Zeitraum von zwei Jahren selbstständig eine gegebene Problematik und Aufgabe bearbeiten und lösen. Angeboten wurde dieses Projekt von der Arbeitsgruppe Betriebssysteme und Verteilte Systeme (AGBS), betreut wurde es von Dr. Jan Bredereke und Dr. Ulrich Hannemann.
Kernthema von TRACS sind sicherheitskritische Systeme (Safety-Critical Systems) insbesondere im Bahnbereich, von deren korrektem Verhalten u.U. Menschenleben abhängen. Dieses Thema erfordert genaue Kenntnisse über sicherheitstechnische Aspekte der Software, Hardware und des Anwendungsgebiets.

1.2 Die Domäne

Der Arbeitsbereich Weltmodell lieferte dem Projekt TRACS eine gemeinsame Grundlage in Bezug auf die Umwelt, in der die von TRACS entwickelten Controller einmal funktionieren sollen, sowie die verwendete Technik. Hier wurden alle Annahmen, Abstraktionen, und Einschränkungen aufgestellt, die für die zu berücksichtigende Umwelt

und die zu steuernde Hardware gestellt werden müssen.
Zur Aufstellung dieser Weltmodelle wurden u.a. Experten der Bremer Straßenbahn AG und der Firma Hanning & Kahl befragt.

1.3 Problembeschreibung

Ziel des Projekts TRACS war die Entwicklung eines wiederverwendbaren, automatisierten und damit Kosten sparenden Bahnsteuerungssystems sowie eines Verifikationsprozesses, mit dem das System auf seine Sicherheit und Korrektheit überprüft werden kann. Das Steuerungssystem besteht dabei aus einem generischen Steuerinterpreter sowie den auf das jeweils zu steuernde Gleisnetz abgestimmten Hardwaretreibern und Projektierungsdaten, welche aus einer domänenspezifischen Beschreibung des Gleisnetzes generiert werden. Der Verifikationsprozess benutzt Methoden des Model-Checkings sowie ausführliche Soft- und Hardware-Tests um sowohl die Sicherheit der Ausgangsdaten als auch die Korrektheit des finalen Steuerungssystems zu gewährleisten. Ein Hauptziel des Projekts war dabei, das System soweit wie möglich automatisiert und wiederverwendbar zu gestalten im Vergleich zu heutigen Systemen, die weitgehend manuell entwickelt und geprüft werden.

1.4 Der Lösungsansatz

Der heute gebräuchlichen Ansatz zur Entwicklung von Gleisnetzsteuerungssystemen sieht in etwa so aus: Aus den vorliegenden Daten eines Gleisnetzes wie Verschluss- und Konflikttabellen oder Routenplänen wird weitgehend manuell das Steuerungssystem hergestellt und implementiert. Das kann den Vorteil haben, dass jeder so entwickelte Controller bestmöglich zu der vorliegenden Situation passt, verlangt aber für jeden Controller erneut einen eigenen Entwicklungsprozess.

Das Projekt TRACS verfolgt einen neuen Ansatz, in welchem das ausführbare Softwaresystem automatisch aus den Anforderungsspezifikationen erzeugt wird. Diese Anforderungsspezifikationen sind die Basis für eine daraus erzeugbare Verifikations- und Testsuite, mit deren Hilfe die Korrektheit von Konfigurationsdaten, ausführbarer Software und der Integration der Software in der Hardware automatisch überprüft werden kann.

1.5 Entwicklungsprozess

Das Projekt TRACS hat einen Entwicklungsprozess ausgearbeitet, in dem das Zusammenspiel aller vom Gesamtsystem benötigten Komponenten festgelegt wird. In diesem Prozess wurden eine Reihe von Komponenten entwickelt, welche zur Erzeugung eines konkreten Bahnsteuerungssystems erforderlich sind. Dies beginnt bereits bei der Eingabe des zu steuernden Gleisnetzes und geht bis zum eigentlichen Controller, der die Steuerung des Systems übernimmt. Das komplette System soll dabei auf einer herkömmlichen PC-Plattform lauffähig sein.
Der Entwicklungsprozess ist außerdem die Grundlage für den Verifikationsprozess.

1.6 Verifikationsprozess

Die Grundlage für den TRACS Verifikationsprozess ist der Entwicklungsprozess. Hier wurde geplant, in welcher Form jeder Zwischenschritt des Entwicklungsprozess verifiziert bzw. validiert werden kann. Der TRACS Verifikationsprozess bildet damit die Grundlage für einen Beweis über die Sicherheit des TRACS Gesamtsystems.

1.7 Sicherheitsnachweis

Ein Bahnsteuerungssystem soll im wesentlichen einen reibungslosen und vor allem sicheren Ablauf des Bahnbetriebs gewährleisten. Um die Sicherheit des von TRACS entwickelten Systems sicherzustellen wurden mögliche Gefahrenquellen identifiziert, welche den Betrieb gefährden könnten. Dazu wurde die Fehlerbaumanalyse als eine systematische Methode verwendet, um alle möglichen Ursachen für Fehlverhalten eines TRACS Controllers zu identifizieren. Sie wird zusätzlich benutzt, um mögliche Fehler im TRACS-internen Entwicklungs- und Verifikationsprozess aufzudecken. Als Ergebnis der Fehlerbaumanalyse erhält das Projekt TRACS eine Liste von Anforderungen an Entwicklungs- und Verifikationsprozess und zu entwickelnde Controller. Da die Fehlerbaumanalyse auf dem TRACS Weltmodell basiert, sind diese resultierenden Anforderungen geeignet, um Verletzungen der Sicherheitsanforderungen innerhalb eines des TRACS Weltmodells entsprechenden Umgebung zu vermeiden.

1.8 Bewertung

Das Ziel, am Ende des Projekts ein Bahnsteuerungssystem zu erhalten ist weitestgehend erreicht - wenn auch nicht alle Details wie ursprünglich angedacht realisiert werden konnten. An der einen oder anderen Stelle gab es Kommunikationsprobleme zwischen den einzelnen Teilgruppen oder aber auch im gesamten Projekt und der Schwund an Projektmitgliedern bis zur Halbzeit des Projekts machte das Erreichen des Ziels nicht gerade einfacher. Letzten Endes steht das System, wobei die angedachte Verifizierung nur teilweise durchgeführt werden konnte.

Henrik Röhrup

Kapitel 2

Die Domäne

2.1 Einleitung

Im folgenden Kapitel wird die Domäne, in der das von TRACS entwickelte System angesiedelt ist, näher beleuchtet. Diese Betrachtung dient als eine wichtige Grundlage der anschließenden Entwicklungsarbeit. Es wird so eine gemeinsame Basis geschaffen, auf der alle Komponenten des Systems aufbauen und an die sie sich halten müssen. Damit wird auch eine mögliche Quelle für Inkonsistenzen zwischen einzelnen Komponenten weitgehend ausgeschaltet.
Das Kapitel ist in insgesamt drei Abschnitte unterteilt:

- **Die Umwelt** - Hier beschreiben wir die Umwelt, in der ein Straßenbahnsystem angesiedelt ist und deren Einfluss es ausgesetzt ist.
- **Die Technik** - Hier beschreiben wir die typischerweise in einem Straßenbahnsystem vorkommende Technik und ihre Funktion.
- **Die Experten** - Hier wird über Kontakte zu Domänenexperten berichtet, die sich während des Projektes ergaben und die zur Informationsgewinnung genutzt wurden.

Arne Stahlbock

2.2 Die Umwelt

Die sichere Fahrt einer Bahn kann durch viele unterschiedliche Aspekte beeinflusst werden, die nicht direkt mit der Bahntechnik zusammenhängen, sondern aus der Umwelt kommen. Als erstes kommt eine Auflistung dieser Aspekte, die dabei bereits in Kategorien zusammengefasst sind, um einen schnelleren Überblick zu liefern und um die Zusammengehörigkeiten der einzelnen Punkte zu verdeutlichen.

2.2.1 Auflistung

Physikalische Wirkungen

- die Bahn direkt betreffend
  - Trägheitskräfte
    - Fliehkräfte
    - Beschleunigung/Verzögerung
    - Gravitation
  - Elektromagnetische Wirkungen
    - Felder um Stromleitungen
    - Handys/Elektronische Geräte
    - Elektrosmog/Funkverkehr
  - Korrosion/Verschleiß
    - Motor
    - Karosserie
    - Steuerkomponenten
    - Elektrik/Batterien
- das Gleisnetz betreffend
  - Korrosion/Verschleiß
    - Gleise verrostet/verschmutzt/spröde
    - Sensoren Kabelbrüche etc.
    - Weichen eingerostet/Kabelbrüche etc.
    - Signale Glühbirnen/Kabelbrüche etc.
    - Stromleitungen
  - Elektromagnetische Wirkungen
    - Falsch beeinflusste Sensoren
  - Hitzewirkungen

Umfeld

- Stromausfälle
- Andere Verkehrsteilnehmer
- Tiere/Hindernisse auf Gleisen

- Geographie
  - Hügel
  - Tunnel
  - Bodenbeschaffenheit
- Gesperrte Teilgleisnetze

Verhalten

- Fahrer
  - Fahrweise
  - Sicht
  - Reaktionszeiten
  - Müdigkeit/Entnervtheit
- Notbremsungen (Passagiere und Fahrer)
- Sehr später Haltewunsch
- Vandalismus/Terror/anderes Fehlverhalten

Wettereinflüsse

- Wind
- Überschwemmung
- Gewitter
- Extreme Temperaturen (Eis/Schnee)
- Eingschränkte Sicht
  - Nebel
  - Starker Regen
  - Hagel
- Feuer

Gesetzliche Einschränkungen

- Mindest-/Maximalgeschwindigkeiten

- Maximale Anzahl Passagiere
- Vorgaben für den Fahrer
- Sicherheitsstandards
- Verkehrsordnung

2.2.2 Textuelle Beschreibung

Als nächstes folgt nun eine kurze Beschreibung der einzelnen Aspekte und auf welche Art und Weise sie die sichere Fahrt der Bahn beeinflussen können. Dabei wird nicht genau auf alle oben genannten Punkte im einzelnen eingegangen, da sich viele von ihrer Beschaffenheit und Auswirkung gleichen oder zumindest sehr ähneln. Andere Punkte stehen wiederum in einem Zusammenhang zueinander, obwohl sie in völlig unterschiedlichen Teilbereichen aufgezählt werden.

2.2.2.1 Physikalische Wirkungen

2.2.2.1.1 die Bahn betreffend Als erstes sollen hier die physikalischen Wirkungen auf die Bahn angesprochen werden.
Hierbei gibt es als erstes die Trägheitskräfte. Von denen gibt es verschiedene Arten.
Die Fliehkräfte wirken sich insbesondere in Kurven aus. Fährt eine Bahn in eine Kurve wird sie durch die Fliehkräfte je nach Geschwindigkeit und Gewicht der Bahn verschieden stark nach außen getragen und könnte in extremen Fällen dadurch entgleisen.
Ebenfalls nach bereits vorhandener Geschwindigkeit und dem Gewicht der Bahn, sowie nach vorhandenem Antrieb kommt es zu unterschiedlichen Beschleunigungen bzw. Verzögerungen, also Bremswegen der Bahn. Hierbei spielen auch noch viele weitere Punkte eine Rolle, wie z.B. die Beschaffenheit der Gleise, die Geographie (fährt die Bahn gerade von einem Hügel runter?), Niederschlag, der sich wiederum auf die Beschaffenheit der Gleise auswirkt usw.
In dem meisten Fällen sollte sich eine geringere Beschleunigung nicht negativ auswirken, es sei denn die Bahn muss schnell einen Gefahrenbereich verlassen.
Problematischer wirkt sich eine geringe Verzögerung aus, was gleichbedeutend mit längeren Bremswegen ist. Kommt es durch die eben genannten Umstände zu einem zu langen Bremsweg kann die Bahn zum Beispiel nicht mehr rechtzeitig an Signalen halten oder sie kommt zu schnell in eine Kurve.
Das kann beim ersten im schwersten Fall zu schweren Unfällen führen, egal ob mit einer anderen Bahn oder sonstigem Verkehr. Im zweiten Fall kann es dann durch die bereits oben erwähnten Fliehkräfte in besonders schlimmen Fällen zu Entgleisungen kommen.
Die Gravitation hat insbesondere Auswirkungen auf die anderen physikalischen Wirkungen, wie die Beschleunigung.
Die Risiken durch diese Einwirkungen lassen sich durch unser System nicht vermindern. Hierbei kommt es vor allem auf den Fahrer an, der seine Fahrweise immer den äußeren Umständen anpassen sollte.
Viel zu steile Kurven sollten zudem bereits beim Gleisbau vermieden werden. Gibt es dennoch gefährlich Bereiche sollten diese durch Geschwindigkeitsbegrenzungen gekenn-

zeichnet sein.

Die elektromagnetischen Wirkungen können auch Einfluss auf die sichere Fahrt der Bahn haben.
Dort gibt es Felder um Stromleitungen, Handys und andere elektronische Geräte und Elektrosmog bzw. Funkverkehr.
All diese Sachen können sich auf die Elektronik in der Bahn auswirken und somit im schlimmsten Fall die Steuerelektronik verwirren. Diese Gefahr sollte aber durch besondere Vorgehensweise bei der Herstellung dieser Elemente ausgeschlossen werden. Genauso wie andersartiger Funkverkehr den Funkverkehr der Bahn stören könnte, diese Frequenzen jedoch reserviert sind.
In diesem Bereich lässt sich unser System auch nicht verbessern.
Physikalische Auswirkungen wie Korrosion und Verschleiß üben sich natürlich auch negativ auf das Fahrverhalten einer Bahn aus.
Dies sind Schäden, die mit dem Alter der Bauteile zusammenhängen. Es ist selbstverständlich, dass bei allen Komponenten im Lauf der Jahre ein Verschleiß entsteht. Besonders schwerwiegend können sich solche Mängel am Motor, der Karroserie, den Steuerkomponenten und der Elektrik auswirken, da hier zum einen die sichere Fahrt direkt gefährdet ist, z.B. durch Ausfälle einzelner oder gar aller dieser Komponenten, aber auch die Passagiere im Innenraum gefährdet werden können.
Auf all diese Punkte hat unser System keinen Einfluß. Hier sollte gute Wartung und Instandhaltung für genügend Sicherheit sorgen.

2.2.2.1.2 das Gleisnetz betreffend Ebenso wie die Bahn, ist auch das Gleisnetz physikalischen Wirkungen ausgesetzt. Dabei kann es sich direkt um die Gleise, wie auch um einzelne der sonstigen Elemente am Gleisnetz handeln. Hierbei kann es ebenso wie bei der Bahn zu Korrosion bzw. Verschleiß kommen.
Dadurch können die Gleise rosten, verschmutzen oder spröde werden, was zu einer erhöhten Gefahr der Entgleisung führt, aber auch den oben besprochenen Bremsweg beeinflusst.
Desweiteren können die Sensoren an der Strecke z.B. Kabelbrüche bekommen, die natürlich die Funktion dieser Sensoren unmöglich machen.
Ähnlich verhält es sich bei den anderen Elementen, wie Weichen und Signalen. Das Auftauchen eines solchen Schadens kann unser System durch fehlende Rückmeldung dieser Komponente bemerken. Desweiteren können noch durch Verschleiß Schäden an den Leitungen entstehen.
Diese Mängel können durch unser System, genauso wie Probleme an den Gleisen, nicht abgefangen werden.

Auch Elemente des Gleisnetzes können durch elektromagnetische Wirkungen beeinflusst werden. Insbesondere könnten hierdurch Sensorwerte verfälscht werden.
Desweiteren gibt es noch Temperatureinwirkungen. Hierbei sind Auswirkungen gemeint die durch Hitze und Kälte entstehen. Also das Ausdehnen und Zusammenziehen von Elementen bei den jeweiligen Temperaturen.
Auf diese zuletzt genannten Auswirkungen kann unser System ebenfalls nicht reagieren. Diese sollten bereits bei der Erstellung des Gleisnetzes bedacht sein.

2.2.2.2 Umfeld

Außer den physikalischen Auswirkungen, gibt es noch Einflüsse durch das Umfeld der Bahn und ihres Gleisnetzes. Hierzu zählen wir alle Einflüsse, die von Außen aus dem Bereich in dem sich die Bahn befindet kommen.
Als erstes seien hier Stromausfälle aufgeführt, die sowohl einen kleinen Bereich als auch das komplette Netz betreffen könnten. Hierbei sollte unser System immer in der Lage sein dies zu erkennen und das gesamte Gleisnetz in einen sicheren Zustand überführen.
Eine weitere Gefahr aus dem Umfeld der Bahn, auf die unser System aber nicht reagieren kann sind andere Verkehrsteilnehmer, wie z.B. Autos, Fahrräder und Fußgänger. Diese können immer plötzlich auf die Gleise kommen. Auf diese Gefahr muss der Fahrer reagieren.
Ein ähnliches Problemfeld sind Tiere oder andere Hindernisse auf den Gleisen (2.3). Auf eine solche Gefahrensituation muss auch der Fahrer reagieren, da unser System ein solches Problem nicht erkennen kann.
Desweiteren ist zum Umfeld die Geographie des Gleisnetzes zu zählen. Dazu zählen wir Hügel, Tunnel und die Bodenbeschaffenheit. Diese können zum Einen auf die Sicht des Fahrers Einfluß haben, sowie auf physikalische Wirkungen, wie dem Bremsweg oder die Beschleunigung. Auf diese Wirkungen kann unser System keinen Einfluss nehmen, hier ist wieder der Fahrer gefragt. Einen weiteren Einfluß auf die Sicht des Fahrers sind die gegebenen Lichtverhältnisse, wobei der Fahrer zum Beispiel geblendet werden kann.
Desweiteren zählen wir zum Umfeld auch gesperrte Teilgleisnetze, da diese kurzfristig durch Unfälle oder Hindernisse gesperrt sein können, aber auch längerfristig durch z.B. Baustellen. Hier sollte eine Umplanung der Routen möglich sein, d.h. es muss möglich sein auf möglichst unkomplizierte Art neue Routen zu integrieren.

2.2.2.3 Verhalten

Sowohl der Fahrer als auch die Passagiere und auch andere Verkehrsteilnehmer unterliegen natürlich der gesamten Spannbreite menschlichen Verhaltens und sind von unserem System nicht weiter beeinflussbar.

2.2.2.4 Wettereinflüsse

Das Wetter kann auch die sichere Fahrt der Bahn beeinflussen. Zumeist passiert dies, indem das Wetter einen der anderen, zuvor beschriebenen Einflüsse auslöst.
Da gibt es zum einen den Wind, der z.B. Bäume zum Umstürzen bringen kann, die dann die Gleise versperren oder gar die Stromleitungen abreissen.
Ein weiterer extremer Wettereinfluß wären da Überschwemmungen, die das Befahren des Gleisnetzes unmöglich machen können.
Extreme Temperaturen können sich auch auf das Gleisnetz auswirken und zusätzlich auch noch dem Fahrer und den Passagieren zu schaffen machen, was dann deren Verhalten beeinflussen kann.
Es gibt aber auch weniger extreme Wetterinflüsse, wie z.B. Gewitter, die Einfluß auf die sichere Fahrt haben können. Durch ein Gewitter kann es zum Beispiel zu Stromausfällen kommen, oder elektronische Elemente können gestört werden.
Desweiteren kann das Wetter auch Einfluß auf die Sicht des Fahrers und Anderer haben. Da kann es Nebel geben oder stark regnen oder hageln. Diese Wetterverhältnisse beeinträchtigen stark die Sicht.
Ein weiterer Punkt der nicht ganz genau den anderen Punkten hier entspricht ist Einfluss durch Feuer. Ein Brand in einer Bahn hat natürlich einen erheblichen Einfluß auf die sichere Fahrt dieser Bahn.
In solchen Fällen kann unser System keinen Einfluß nehmen. Dies trifft auf fast alle bisher genannten Einwirkungen zu. Bei einigen wenigen davon kann unser System bei der Erkennung helfen und damit eine schnellere Behebung ermöglichen (Hardwarestörungen).
Bei den meisten der oben genannten Fälle sollte unser System nach Erkennung auch von Außen schnell das Gleisnetz in einen sicheren Zustand überführen.

2.3 Die Technik

2.3.1 Überblick

Nachdem nun die Umwelt beschrieben wurde, kommen wir in diesem Abschnitt zum zweiten für unser Weltmodell bedeutenden Teil, nämlich der Bahntechnik. Im Folgenden beschreiben wir die Bahntechnik, mit der zusammen unser Steuersystem eingesetzt werden soll, und ihre Funktionsweise. Darüber hinaus zeigen wir, wo wir Vereinfachungen vornehmen (beispielsweise nur einen Teil der eigentlich zur Verfügung stehenden Funktionalität nutzen) und wo wir Bedingungen und Anforderungen an die Technik stellen. Damit wird eine Basis geschaffen, auf der die Entwicklung aller Systemkomponenten aufgebaut wird und somit die Gefahr von Inkonsistenzen zwischen einzelnen Komponenten verringert. Die hier vorliegenden Beschreibungen sind bindend für alle im Projekt entwickelten Systemkomponenten.

2.3.2 Gleisnetzelemente

Gleisnetze bestehen für unser System aus folgenden Elementen:

- Gleise
  - Geraden
  - Kurven
  - Weichen
  - Kreuzungen
- Sensoren
- Signale
- Bahnen
- Verkehrszeichen

2.3.2.1 Gleise

Gleise werden von Bahnen befahren. Ein komplettes Gleisnetz setzt sich dabei aus einer Vielzahl von Gleisstücken zusammen. Diese Gleisstücke sind derart miteinander verbunden, dass eine Bahn von einem Gleisstück auf ein benachbartes Gleisstück fahren kann (Normalfall). Gleisstücke können aber auch beschädigt sein, was zum Entgleisen einer Bahn führen kann. Eine solche Beschädigung muss nicht zwangsläufig für das Steuersystem oder für einen Betrachter sichtbar sein, sondern kann unvorhergesehen auftreten

und zum Entgleisen führen. Ein Gleis ist passiv, es kann vom Steuersystem kein direkter Einfluss auf das Gleis ausgeübt werden. Ein Gleisstück hat eine Länge, was im Zusammenspiel mit der Länge der darauf fahrenden Bahnen eine Maximalzahl von Bahnen ergibt, die sich gleichzeitig ohne zu kollidieren auf diesem Stück befinden können.

2.3.2.1.1 Verknüpfungen von Gleisstücken Eine Gerade oder eine Kurve hat zwei Enden. Eine Weiche hat auf der einen Seite ein Ende, auf der anderen Seite zwei oder drei Enden. Eine Kreuzung hat vier Enden, wobei jeweils zwei Enden einander zugehörig sind. An einem Ende eines Gleisstücks kann ein weiteres Gleisstück angeschlossen sein, so dass eine Bahn hier vom einen auf das andere Gleisstück fahren kann. Ist an einem Ende kein Gleisstück angeschlossen, endet hier das Gleisnetz. An solchen Stellen können Bahnen in das Netz gelangen oder aus ihm entfernt werden. Eine weitere Möglichkeit ist, dass an solchen Stellen das Gleisnetz mit einem Prellbock abgeschlossen wird, so dass kein Einfahren oder Verlassen möglich ist.
Für alle Gleiselemente gilt, dass es zu Kollisionen zwischen Bahnen kommen kann, wenn sich a) mehrere Bahnen gleichzeitig auf dem Element befinden und es von unterschiedlichen Enden her befahren oder b) wenn von zwei hintereinander fahrenden Bahnen die zweite schneller fährt.
Eine Bahn kann sich gleichzeitig auf mehreren Gleiselementen befinden. Sie erstreckt sich dann gerade über einen oder mehrere Verknüpfungspunkte zwischen Gleiselementen.

2.3.2.1.2 Eigenschaften der verschiedenen Gleisformen

- **Gerade** Eine Gerade ist ein kreuzungsfreies Gleisstück mit zwei Enden. Eine auf einer Geraden fahrenden Bahn kann nicht mit einer Bahn, die sich auf einem anderen Gleisstück befindet, das nicht an einem Ende dieser Geraden angeschlossen ist, kollidieren. Sind zwei nicht miteinander verbundene Gleisstücke räumlich dennoch so nah zusammenliegend, dass es zu Kollisionen zwischen Bahnen auf dem einen und dem anderen Stück kommen kann, so werden diese eigentlich getrennten Elemente als ein Element, nämlich eine Kreuzung, betrachtet. Auf einer Geraden gilt jeweils eine Höchstgeschwindigkeit.

- **Kurve** Eine Kurve ähnelt der Geraden, nur dass ihr Verlauf nicht gerade ist. Sie hat neben ihrer Länge demnach eine weitere Kenngröße, den Kurvenradius. Es kann in Kurven in Abhängigkeit von Kurvenradius und Geschwindigkeit einer Bahn zu Entgleisungen kommen. Folglich gilt eine physikalisch mögliche Höchstgeschwindigkeit in jeder Kurve. Es kann jedoch nicht davon ausgegangen werden, dass eine dem Fahrer vorgeschriebene maximale Fahrgeschwindigkeit in jeder Situation sicher ist, diese kann auch über der physikalisch möglichen Höchstgeschwindigkeit liegen. Der Fahrer muss hier also auch eigenständig sicherstellen, in Kurven nicht wegen zu schneller Fahrt zu entgleisen.

- **Kreuzung** Eine Kreuzung hat vier Enden, wobei jeweils zwei Enden einander zugeordnet sind. Es lassen sich somit zwei Strecken bestimmen, die sich genau wie die oben beschriebenen Geraden oder Kurven verhalten. Eine an einem Ende einfahrende Bahn wird bei stetiger Vorwärtsfahrt die Kreuzung an dem zugeordneten Ende wieder verlassen. Als Kenngrößen einer Kreuzung gibt es zwei Längen, nämlich die der beiden Strecken. Auf Kreuzungen kann es durch überhöhte Geschwindigkeit zu Entgleisungen kommen, insofern ist eine Maximalgeschwindigkeit vorgegeben, die jedoch für jede Richtung unterschiedlich sein kann.

  Darüber hinaus gibt es Kreuzungsweichen; diese werden, da sie zu den schaltbaren Elementen zählen, im folgenden Abschnitt über Weichen erklärt.

- **Weiche** Eine einfache Weiche hat auf der sogenannten spitzen Seite (stem) ein Ende, auf der stumpfen Seite (branch) zwei oder drei. Die Weiche kann dabei verschiedene Verbindungen herstellen, das Ende der spitzen Seite kann mit einem der Enden der stumpfen Seite verbunden sein. Folglich sind mehrere Strecken über eine Weiche möglich, zu einem Zeitpunkt jedoch immer maximal eine. Es ist ebenso möglich, dass zu einem Zeitpunkt gerade keine Verbindung geschaltet ist. Eine am spitzen Ende der Weiche einfahrende Bahn wird bei fortwährender Vorwärtsfahrt an das gerade aktiv geschaltete stumpfe Ende gelangen. Ist gerade keine Verbindung geschaltet, kann es zu Entgleisungen kommen. Bei Einfahrt von einem stumpfen Ende sind verschiedene Fälle zu unterscheiden:

  - Einfahrt von dem gerade aktiv geschalteten Ende führt die Bahn zum spitzen Ende
  - Einfahrt von einem nicht aktiv geschalteten Ende kann
    * zur Beschädigung der Weiche führen, wenn sie die Eigenschaft „nicht auffahrbar“ hat
    * die Weiche auf das stumpfe Ende, von dem die Einfahrt erfolgt, umschalten lassen, so dass die Bahn die Weiche überfahren kann und an das spitze Ende gelangt
    * wie vor, und zusätzlich schaltet die Weiche nach Überfahren wieder in die ursprüngliche Stellung zurück, wenn sie vom Typ „Rückfallweiche“ ist.

  Schaltet eine Weiche gerade um, während sie befahren wird (abgesehen von dem Umschalten durch Auffahren an Weichen, die auffahrbar sind), kann es zu Entgleisungen kommen. Als Schaltzustände einer Weiche können demnach die möglichen schaltbaren Strecken benannt werden sowie der Zustand „gerade keine Strecke geschaltet“. Es gibt jedoch eine Vorschrift, die besagt, dass Weichen während der

Anwesenheit einer Bahn nicht umschalten dürfen (und dagegen eigenständig abgesichert sein müssen), man darf an dieser Stelle aber nicht zwangsläufig davon ausgehen, dass diese weltweit gilt.

Darüber hinaus können Weichen defekt sein, in diesem Fall kann es zu Entgleisungen kommen, wenn eine Bahn eine defekte Weiche befährt.

Eine Weiche kann eine automatische Schaltvorrichtung haben, die auf Anforderung eine Schaltung vornimmt. Als Kenngröße ist hierbei die maximale Schaltzeit zu nennen - ist nach dieser Zeit keine Schaltung erfolgt, muss die Weiche als defekt angesehen werden. Eine Weiche kann aber auch immer unter Einsatz einer Metallstange manuell umgeschaltet werden. Eine Weiche ohne Schaltvorrichtung, die also nur manuell zu schalten ist, wird als Passivweichen bezeichnet. Das Verhalten einer Weiche für den Fall, dass während der Umsetzung eines Schaltbefehls (die Weiche ist also im Zustand „gerade keine Strecke geschaltet“) ein weiterer Befehl eingeht, ist nicht definiert.

Kenngrößen einer Weiche sind in jedem Fall die Längen der über sie möglichen Strecken, optional auch die o.a. Schaltzeit, wenn eine Schaltvorrichtung vorhanden ist. Eine Weiche mit Schaltvorrichtung kann abgeschaltet werden, so dass sie in diesem Fall nur noch wie eine Passivweiche arbeitet. Eine Weiche meldet ihren aktuellen Zustand an das Steuersystem. Auch eine Weichen gilt jeweils eine Maximalgeschwindigkeit, die jedoch für jede Richtung unterschiedlich sein kann.

Es gibt weiterhin Kreuzungsweichen. Diese bestehen aus einer Kreuzung, bei der zusätzlich zwei (Einfachkreuzungsweiche, EKW) oder vier (Doppelkreuzungsweiche, DKW) Weichen so angebracht sind, dass an dieser Kreuzung auch abgebogen werden kann. Dabei ist von einem der beiden kreuzenden Gleise nur Rechtsabbiegen, vom anderen nur Linksabbiegen möglich. Bei der EKW besteht diese Abbiegemöglichkeit nur von je einer Seite, bei der DKW von beiden Seiten. Das gesamte Element ist so kompakt gebaut, dass es immer nur von einer Bahn gleichzeitig befahren werden darf. In diesem Sinne unterscheidet es sich von einer Kreuzung, bei der Abbiegemöglichkeiten in größerem Abstand zum Kreuzungspunkt existieren. Dort könnten ggf. mehrere Bahnen gleichzeitig abbiegen. Zur Ansteuerung der „Teilweichen“ der Kreuzungsweiche werden zwei Varianten berücksichtigt:

- individuelle Stellung der Teilweichen
- Kopplung der Teilweichen zu einem Gesamtsystem, das nur zwei Stellungen hat: alle Teilweichen auf „geradeaus“ oder alle auf „abbiegen“

2.3.2.2 Sensoren

Ein Sensor ist ein passives Element, das sich auf/an einem Gleis befindet. Er kann also einem bestimmten Punkt auf dem Gleis zugeordnet werden. Sensoren haben die

Funktion, dass sie, wenn eine Bahn über sie hinwegfährt, dem Steuersystem über diesen Vorgang Auskunft geben. Übermittelt werden können

- die Tatsache, dass sie überfahren wurden (Toggle) oder
- die Tatsache, dass zum aktuellen Zeitpunkt eine Bahn auf dem Sensor anwesend ist (State)

Zusätzlich zu einer dieser beiden Optionen können Sensoren über Richtungserkennung verfügen, also melden, in welche Richtung sie überfahren wurden.
Als weiteren Typ gibt es auch noch Route-Request-Sensoren. Diese dienen den Bahnen zum Anfordern einer Route durch das Gleisnetz. Sie werden nicht von jeder sie überfahrenden Bahn ausgelöst, sondern nur von denjenigen, die die Route befahren wollen, welche der Sensor verwaltet. Diese Information, welche Route sie befahren will, muss von der Bahn übermittelt werden (bspw. Funk). Über weitere Funktionalität verfügt ein Route-Request-Sensor nicht.
Für alle Typen gilt, dass sie defekt sein können, wobei ein Defekt nicht zwangsläufig vom Steuersystem zu erkennen ist. Ebenso können sie abgeschaltet werden. Jeder Sensor hat eine Verzögerung, die zwischen tatsächlicher Auslösung durch eine Bahn und dem Zeitpunkt der Rückmeldung an das Steuersystem gilt.

2.3.2.3 Signale

Ein Signal ist ein Element, das sich räumlich nicht auf dem Gleis, sondern nur in dessen Nähe befindet. Es ist jedoch genau wie ein Sensor einem bestimmten Punkt auf dem Gleis zugeordnet. Ein Signal ist ein Anzeigeelement, das dem Fahrer Anweisungen gibt. Mögliche Anweisungen sind:

- Fahrt nicht freigegeben (Stop)
- Fahrt geradeaus freigegeben (Go straight)
- Fahrt nach rechts freigegeben (Go right)
- Fahrt nach links freigegeben (Go left)
- Warten geradeaus (Wait straight)
- Warten rechts (Wait right)
- Warten links (Wait left)

Ein Signal verfügt immer über die Möglichkeit Stop zu zeigen sowie über eine der drei Go-Varianten. Es kann zusätzlich weitere Go-Varianten und/oder Warten-Varianten haben, wobei Warten-Varianten nur für diejenigen Richtungen vorhanden sein können, für die

auch ein Go existiert. Ein Signal mit zwei Schaltmöglichkeiten wechselt immer zwischen diesen zwei Zuständen. Signale mit Wait-Anzeige können diesen Zustand zwischen dem Stop-Zustand und dem der Wait-Richtung entsprechenden Go-Zustand annehmen. Die Abfolge ist also Stop - Wait(für eine Richtung) - Go(für diese Richtung) - Wait(für diese Richtung) - Stop. Wait ist also wie Gelb bei einer normalen Verkehrsampel zu verstehen. Von allen Wait- und Go-Anzeigen eines Signals darf zu einem Zeitpunkt maximal eine aktiv sein. (Da ein Signal für mehrere Richtungen vor einer Weiche steht und diese Weiche nur in eine Richtung gleichzeitig geschaltet sein kann, macht alles andere auch keinen Sinn - selbst wenn also auch andere Zustände prinzipiell anzeigbar wären, darf das Steuersystem sie nicht erreichen.) Ist für eine Richtung Wait oder Go angegeben, bedeutet dies für alle anderen Richtungen automatisch Stop.
Ein Signal kann optional außerdem über mehrere Anzeigezustände „A“ verfügen. Diese bedeuten „Anforderung eingegangen“ und signalisieren dem Fahrer, dass seine Routenanforderung im Steuersystem angekommen ist und umgesetzt werden wird. Ein Zustand „A“ wird dabei parallel zu einem der übrigen Zustände (Stop, Go, Wait) eingenommen. Die verschiedenen „A“-Zustände korrespondieren mit den Richtungen, die das Signal freigeben kann, ein Signal, das „Go right“ und „Go left“ anzeigen kann, könnte entsprechend auch „A right“ und „A left“ haben, um anzuzeigen, welche Richtung angefordert worden ist. Physisch verfügt das Signal über entsprechend viele „A“-Leuchten, die entsprechend ihrer Bedeutung (left, straight, right) nebeneinander angeordnet sind. Es muss nicht für jede Richtung, die das Signal freigeben kann, ein „A“ vorhanden sein, umgekehrt aber darf kein „A“ für eine nicht vorhandene Richtung existieren. Ein typischer Ablauf wäre folgender: Eine Bahn überfährt einen Route-Request-Sensor und fordert eine Route an. Anschließend nähert sie sich dem Signal, das ihre Route freigeben wird. Dieses steht auf „Stop“ und, nach Eingang der Anforderung, zusätzlich auf „A“. Sobald die Route geschaltet ist, wird „Go“ gegeben, die Bahn fährt in die Route ein. Sobald sie den nächsten Sensor innerhalb der Route erreicht, erkennt das Steuersystem, dass die Bahn ihre Route begonnen hat, setzt das Signal wieder auf „Stop“ und schaltet gleichzeitig das „A“ ab.
Ein Signal schaltet sich auf Anforderung des Steuersystems um, wobei eine maximale Schaltzeit wie bei der Weiche existiert. Das Verhalten eines Signals für den Fall, dass während der Umsetzung eines Schaltbefehls ein weiterer Befehl eingeht, ist nicht definiert.
Ein Signal kann defekt sein, dieser Zustand muss nicht zwangsläufig vom Steuersystem erkennbar sein. Weiterhin sind Signale abschaltbar. Signale melden ihren aktuellen Zustand an das Steuersystem.

2.3.2.4 Bahnen

Straßenbahnen befahren das Gleisnetz. Kenngrößen einer Bahn sind ihre räumlichen Dimensionen, ihre Höchstgeschwindigkeit und ihr maximaler Bremsweg für verschiedene,

im Betrieb mögliche Geschwindigkeiten (maximal bedeutet in diesem Zusammenhang: unter ungünstigsten noch für Fahrtbetrieb zugelassenen Umweltbedingungen, aber bei voll funktionsfähiger Bahn). Gesteuert werden die Bahnen von einem menschlichen Fahrer. Dieser reagiert auf den Schienenverkehr, die Verkehrszeichen und auf die Signale. Der Aufenthaltsort von Bahnen im Gleisnetz wird vom Steuersystem mittels der Sensorenmeldungen erfasst. Darüber hinaus sind dem Steuersystem ggf. angeforderte Routen der Bahn bekannt.

2.3.2.5 Verkehrszeichen

Verkehrszeichen sind ebenso wie Signale nahe der Strecke befindliche Anzeigen, die Anweisungen an den Fahrer darstellen. Im Unterschied zum Signal sind ihre Anzeigen aber nicht veränderbar, sondern statisch. Das Steuersystem kann über Verkahrszeichen demnach keinen Einfluss ausüben.

2.3.3 Steuersystem

Beim Steuersystem handelt es sich um einen Rechner, auf dem die von uns entwickelte Steuersoftware läuft. Dieses Programm nimmt Meldungen der Sensoren, Weichen und Signale entgegen und „entscheidet“ auf Basis dieser Meldungen und der Gleisnetzdaten, welche Weichen und welche Signale wann wie zu schalten sind. Das System soll sicherstellen, dass

- keine Kollisionen zwischen Bahnen auftreten
- jede Bahn die von ihr gewünschte Route zeitnah befahren kann
- keine Fahrten freigegeben werden, die zur Beschädigung von Bahntechnik führen können

Zum Steuersystem zählen weiterhin die Hardwarecontroller, die den Weichen, Signalen und Sensoren zugehörig sind sowie menschliche Bediener.
Im Prinzip ist das Steuersystem ebenfalls ein Element der Bahntechnik, da aber Teile des Steuersystems von uns entwickelt werden, während die übrige Bahntechnik bereits wie beschrieben vorliegt, wird es hier gesondert aufgeführt. Zudem kann ein Gleisnetz auch ohne Steuersystem betrieben werden, aber umgekehrt hat ein Steuersystem ohne Gleisnetz keinen Nutzen.

2.3.4 Gleisnetzdaten

Nicht direkt als „Bahntechnik“ zu bezeichnen, dennoch in diesem Zusammenhang aufzuführen sind verschiedene Dokumentationen über das Gleisnetz, die insbesondere für den Betrieb des Steuersystems benötigt werden. Dazu zählen:

- Verschlusstabellen:
  - Routendefinitionstabelle: definiert die durch das Gleisnetz zu befahrenden Routen, das müssen nicht zwangsläufig alle überhaupt möglichen Routen sein, sondern nur die vom Gleisnetzbetreiber gewünschten
  - Konflikttabelle: zeigt auf, welche Routen miteinander in Konflikt stehen und daher nicht zeitgleich befahren werden dürfen
  - Weichenstellungstabelle und Signalstellungstabelle: zeigen auf, welche Weichen- bzw. Signalstellungen jeweils für das Befahren der definierten Routen benötigt werden
- Gleisnetzbeschreibung: Dokumentiert, wie das Gleisnetz als solches aus den in diesem Weltmodell beschriebenen Elementen zusammengesetzt ist. Diese kann u.a. als sogenannter Netzgraph, einer CAD-Zeichnung, vorliegen.
- Hardwarebeschreibung: Dokumentiert die verschiedenen im Gleisnetz verwendeten Hardwareelemente und die für die Steuerung relevanten Eigenschaften. Jedem mit dem Steuersystem in Verbindung stehenden Element (Sensoren, Weichen, Signale) muss der genaue Hardwaretyp sowie dessen Schaltzeit (Signale, Weichen) bzw. Reaktionszeit (Sensoren) zugeordnet sein.

Zum oben verwendeten Routenbegriff folgende Definition: Als Route wird eine zusammenhängende Folge von Gleisstücken betrachtet, die von zwei Sensoren begrenzt wird (Eingangs- und Ausgangssensor) und vor der sich ein Einfahrtssignal befindet. Eine Bahn kann demnach eine Route in einem Zug durchfahren. Ein Gleisstück kann zu mehreren Routen gehören. Wenn dieser Fall auftritt, stehen die über dieses Gleisstück führenden Routen miteinander in Konflikt.

2.3.5 Annahmen

An ein von unserem System zu steuerndes Gleisnetz werden bestimmte Bedingungen gestellt. Diese werden nun aufgeführt.
In Bezug auf Sensoren und Signale werden an das Gleisnetz folgende Bedingungen gestellt:

- Alle Routen, die dasselbe Einfahrtssignal haben, müssen von demselben Route-Request-Sensor (RR-Sensor) verwaltet werden (oder von einem jeweils eigenen, die dann aber so nah zusammenliegen müssen, dass sie als Gesamtheit die u.a. Bedingung an die Elementabfolge und -distanz für den Beginn einer Route erfüllen).
- Jede Route darf nur von einem RR-Sensor verwaltet werden

- Der Ausfahrtssensor einer Route muss so weit von der letzten Kreuzung oder Weiche entfernt sein, dass eine ihn auslösende Bahn zum Auslösezeitpunkt garantiert diese Kreuzung oder Weiche verlassen hat (Distanz Kreuzung/Weiche –> Sensor ist größer als größte Länge der eingesetzten Bahnen).
- Die Abfolge von Elementen am Beginn einer Route muss wie folgt sein: RR-Sensor (oder RR-Sensorengruppe), Einfahrtssignal, Einfahrtssensor. Diese Elemente müssen so nah zusammenliegen, dass von dem RR-Sensors / der RR-Sensorengruppe bis zum Einfahrtssensor maximal drei Bahnen Platz haben, so dass eine nachfolgende vierte Bahn erst einen RR-Sensor auslösen kann, wenn die erste Bahn den Einfahrtssensor berührt hat und somit in das Gleisnetz eingetreten ist. (Es können also niemals mehr als drei Requests an einer RR-Gruppe gleichzeitig anliegen.)

An das Verhalten eines Bahnfahrers werden folgende Forderungen gestellt:

- Fährt er hinter einer anderen Bahn her, darf er nicht auffahren.
- Signale sind zu beachten, sofern dies noch möglich ist (wird ein Signal also auf Stop gesetzt, während die Bahn dem Signal schon näher ist als der o.a. maximale Bremsweg, darf nicht zwangsläufig erwartet werden, dass die Bahn noch vor dem Signal halten kann).
- Verkehrszeichen sind zu beachten.
- Erkennt er einen Defekt eines Gleiselementes, so darf er dieses nicht befahren.
- Erkennt er einen Defekt eines Signals, so darf er daran nicht vorbeifahren, ohne dass ihm dies von anderer Stelle, bspw. Betriebszentrale, genehmigt wird
- An Haltestellen ist anzuhalten, wenn Fahrgäste ein- oder aussteigen wollen.
- Beim Auftreten von Hindernissen auf dem Gleis ist zu halten.
- Es darf in den Bereichen, die vom Steuersystem kontrolliert werden, nicht rückwärts gefahren werden, solange das Steuersystem aktiv ist.

2.3.6 Abstraktionen

Da es in der Welt viele Gleisnetze geben mag, die nicht von obiger Beschreibung abgedeckt werden, die aber dennoch von unserem System beherrschbar sein sollen, nehmen wir an einigen Stellen Abstraktionen vor, um das Verhalten bzw. die Eigenschaften komplizierterer Elemente und Elementgruppierungen auf einfachere Fälle zurückzuführen. Diese listen wir an dieser Stelle auf.

- Gleisverschlingungen (Stellen, an denen ein Strang eines Gleises sich zwischen den beiden Strängen eines anderen Gleises befindet) werden als Kreuzungen interpretiert
- Bereiche, in denen zwei Gleise zwar nicht ineinander verschlungen sind, aber sich Bahnen dennoch nicht kollisionsfrei begegnen können werden ebenfalls als Kreuzungen betrachtet
- Vorsignale werden vom geplanten Steuerungssystem nicht berücksichtigt
- Signale zur Vorgabe von Höchstgeschwindigkeiten - werden lediglich als Signale zur Freigabe der Fahrt interpretiert
- Signale zur Anzeige des Weichenzustandes - werden vom Steuerungssystem nicht direkt angesteuert, können aber an Weichen gekoppelt sein und die Daten zur Anzeige direkt von diesen beziehen
- Signalzustand „T“ (häufig an Haltestellen anzutreffen) - wird von uns mit dem „Wait“-Zustand gleichgesetzt (d.h. anschließend kommt Go)
- Weitere evtl. existierende Signale werden nicht berücksichtigt
- Haltestellen - ein zwischen zwei Sensoren durchgeführter Halt würde für das Steuersystem, das lediglich die Sensorendaten bekommt, genauso aussehen wie eine Langsamfahrt zwischen diesen Sensoren; da Langsamfahrt grundsätzlich möglich ist, ist das Halten (an Haltestellen oder anderswo) damit bereits abgedeckt
- Vorsortierweichen (Weichen, an denen nach der Aufspaltung in zwei Gleise diese beiden Gleise noch über eine längere Strecke verschlungen sind) werden als normale Weichen mit längeren Zweigen angesehen
- Routen, die zwischen Eingangs- und Ausgangssensor noch weitere Sensoren haben, werden ohne diese zusätzlichen Sensoren betrachtet (deren Informationen also nicht ausgewertet)

Arne Stahlbock

2.4 Die Experten

2.4.1 Überblick

Zur Aufstellung der vorher beschriebenen Weltmodelle musste sich das Projekt viel Wissen über die Domäne aneignen. Hierzu wurde in der Literatur geforscht, aber auch Kontakt zu Firmen, die in der Domäne tätig sind, aufgenommen. Relativ früh in der

Projektzeit wurde Kontakt zu der Bremer Straßenbahn AG aufgenommen und dann ein Treffen mit Vertretern vereinbart, um genauere Informationen zu der Domäne zu erhalten. Auf Basis dieser Informationen konnte man dann mit den Weltmodellen beginnen. Im späteren Verlauf ergab sich dann noch ein Treffen mit der Firma Hanning & Kahl. Hierbei wurde erhofft eine Validierung unserer Weltmodelle zu erhalten. Genaueres zu den Treffen folgt.

Waldemar Wockenfuß

2.4.2 Bremer Straßenbahn AG

2.4.2.1 Überblick

Anlass zur Kontaktaufnahme mit der Bremer Straßenbahn AG
Die seit Beginn des Projektes TRACS im Oktober 2003 zur Verfügung stehenden Informationen hatten sich teilweise als zu abstrakt und theoretisch erwiesen, um ein praktisches Verständnis für die Thematik zu ermöglichen. Um einen praktischen Einblick über aktuell eingesetzte Bahnsteuerungssysteme eines lokalen Nahverkehrdienstleisters, der Bremer Straßenbahn AG (BSAG), zu erhalten, erfolgte erstmals Mitte November auf Einzelinitiative hin Kontaktaufnahme mit dieser. Aus diesen Gründen erfolgte dir Gründung der Arbeitsgruppe BSAG, um sich um den Kontakt zu der Bremer Straßenbahn AG zu kümmern.

Arbeitsziele
Zunächst sollte ein Besuchstermin ausgehandelt werden und eine allgemeine Vorstellung des Projektes TRACS erarbeitet und im Rahmen des Besuches vorgestellt werden. Auch die Sammlung eines Fragenkatalogs gehörte hierzu. Im Plenum vom 30.01.2004 wurde zusätzlich vereinbart, dass jede Arbeitsgruppe (DSL, HW/SW-Integration, Modelchecking und Steuerinterpreter) einen Kurzvortrag über ihren Teilbereich erstellt, von der AG-BSAG in den Gesamtvortrag einarbeiten lässt und beim Besuch vorträgt.

Besuch der Ausstellung „Das Depot“
Die Straßenbahn-Ausstellung „Das Depot“ ist eine Ausstellung, die von ehrenamtlichen Mitarbeitern geleitet wird: die „Freunde der Bremer Straßenbahn e.V.“. Wir entschlossen uns zu einem Besuch der Ausstellung, da sich das Treffen mir der BSAG immer weiter nach hinten verschob und wir einige Fragen bezüglich des Straßenbahnsystems beantwortet haben wollten, um etwas weiterarbeiten zu können. Diese Fragen bezogen sich auf die grundlegenden Konzepte des Straßenbahnbetriebs.

Anfang März 2004 trafen wir uns mit den ehrenamtlichen Mitarbeitern vom „Depot“ und erhielten eine persönliche Führung, bei der uns alle Fragen, die wir stellten, gern und nach besten Wissen beantwortet wurden. Dass dieser Besuch nicht das Treffen mit der

BSAG ersetzen würde, war von vornherein klar. Wir erhielten viele interessante Informationen, vor allem über Weichen, Signale, Schilder und Ampelsteuerung. Die Mitarbeiter vom „Depot“ stellten uns den Straßenbahnbetrieb aus Sicht des Straßenbahn-Fahrers vor. Bei unseren tiefergehenden Fragen zu den technischen Gegebenheiten konnten sie uns leider nicht viel weiterhelfen.
Für genauere Informationen unseres Besuchs liegt eine Mitschrift der Ausstellung vor, welche in elektronischer Form auf der Webseite unseres Projekts zu finden ist. Für weiterführende Informationen zur Ausstellung sei auf die Internetseite der Ausstellung [Bre04] und auf den Webauftritt der „Freunde der Bremer Straßenbahn e.V.“ [Fre04] verwiesen.

2.4.2.2 Arbeitsschritte

Vortrag
Nach Festlegung der Gruppenmitglieder wurde ein Vortrag ausgearbeitet und gegen Ende des ersten Semesters eine erste Version an alle Projektmitglieder gesendet. Auf dem Projektwochenende wurde ein kurzer Überblick des Vortrags zur Diskussion gebracht und festgestellt, dass dieser auch für Besuche bei weiteren Firmen genutzt werden soll. Nach Einarbeitung einiger Anregungen wurde eine neue Version zur Durchsicht verschickt. Wiederum wurden Anregungen, nun mehr das Layout als Inhaltliches eingearbeitet, zusätzlich eine Grafik als Projektübersicht sowie die beschlossenen Kurzvorträge der einzelnen Gruppen.
Nach mehreren Revisionen wurde dann eine entgültige Fassung erst kurz vor Beginn des zweiten Semesters fertig.
Für den Fragenkatalog wurden der BSAG-Gruppe aus den anderen Gruppen Fragen gestellt, die sich teilweise aufgrund der Informationen von Prof. Dr. Jan Peleska erledigten, wohingegen natürlich auch neue Fragen auftauchten.

Besuchsorganisation
Die erste Kontaktaufnahme erfolgte im November 2003 per Telefon. Es gab ein Gespräch mit Herrn Kai Teepe (Bauleitung), bei dem kurz das Projekt TRACS und die uns interessierenden Themen vorgestellt wurden. Da Herr Teepe verantwortlich für die Bauarbeiten am Schienennetz war, wurde als Besuchstermin statt vor Weihnachten 2003 Ende Januar 2004 vereinbart. Bei der erneuten Kontaktaufnahme Mitte Januar 2004 wurden wir nach erneuter Kurzvorstellung des Projektes TRACS weiterverwiesen an Herrn Herrmann Lübbers, der aber leider bis Mitte Februar telefonisch nicht zu erreichen war. Herrn Lübbers wurde das Projekt nochmal vorgestellt und eine Themenliste sowie der Fragenkatalog gegeben,woraufhin der Besuchstermin bei der BSAG für den 10.03.2004 vereinbart wurde.

Besuchsablauf

Der Besuch fand statt im BSAG-Zentrum Neustadt, wo wir von dem bereits genannten Herrn Lübbers, Bereich Schienenfahrzeugtechnik, und Herrn Hatesaul, Bereich Rechnergestütztes Betriebsleitsystem, empfangen wurden. Nach einer kurzen Vorstellung wurde der Vortrag, unterbrochen von Zwischenfragen, präsentiert. Herr Lübbers und Herr Hatesaul hielten ihrerseits drei Vorträge, zum einen über das neue Zugsteuerungssystem IMU-100 von Siemens, das bis Ende des Jahres das bisherige IMU-94 ablösen soll, zum anderen über ihre rechnergestützte Betriebsführung. Anschliessend wurden die Fragen aus dem Fragenkatalog gestellt und größtenteils beantwortet.

Präsentation des Projektes TRACS
Die Zwischenfragen beim Vortrag machten deutlich, dass die Definitionen doch sehr unterschiedlich ausfallen. So mussten Begriffe wie Verschlusstabellen (Interlocking tables), besonders aber der Umfang unserer Routen (kleinere Gleisabschnitte mit wenigen Weichen statt die komplette Linie 6 der BSAG vom Flughafen zur Uni) und die Größe unseres Gleisnetzes (gesamtes Straßenbahnnetz einer Stadt) erklärt werden. Auch musste unser Aufgabenbereich insofern eingegrenzt werden, als dass wir uns nur um die sichere Steuerung der Straßenbahnen selber kümmern, alle anderen Verkehrsteilnehmer aber außer Acht und somit der Verantwortung des Fahrers überlassen.
Darüberhinaus sagten wir zu, auf vorgefertigte Schnittstellen zur HW zurückgreifen zu wollen, was zu dem Zeitpunkt in Frage stand.

Vortrag von Herrn Hatesaul zur Schienen-/Fahrzeugtechnik und Beantwortung des Fragenkatalogs
Generell hat jede Straßenbahn einen Joystick, mit dem der Fahrer über eine Sendespule im Bordcomputer (Ibis Wagenbus) und eine Empfangsspule Weichen nach rechts, links oder geradeaus gesteuert werden können. Die Lichtsignalanlagen werden durch Berührung von Kontakten an den Oberleitungen geschaltet. Hier greift kein zentrales System ein, jede Weiche lässt sich nur dann bedienen, wenn keine andere Straßenbahn auf ihr steht oder sie schon gestellt hat. Die Eingabe der Routen geschieht derzeit noch per Datenfunk, nach der Umstellung auf das neue Zugsteuerungssystem werden aber alle Daten morgens per CD in den Bordcomputer gespielt, so dass die Weichen automatisch gestellt werden, während der Fahrer weiterhin jederzeit manuell steuern kann und so auch weiterhin die Verantwortung trägt. So wird auch keine automatische Zwangsbremsung ausgelöst. Der Befehl, eine Weiche zu stellen, wird in dem sogenannten Weichentelegramm erteilt, dass etliche Informationen wie Routenziel, Linie, Routennummer, Kurs, Wagennummer, den Richtungswunsch und eine Prüfnummer enthält (CRC), und muß dreimal in Folge erteilt werden.
Die Signalsteuerung ist unabhänging von der der Weichen. Die älteren, noch mit Glühbirnen ausgestatteten Anlagen werden durch solche mit LEDs ersetzt, da es inzwischen auch Baugruppen zur Ausfallerkennung dieser gibt. In so einem Fall sperrt die Weiche, so dass der nächste Fahrer die Zentrale informieren und dann durch manuelles Verstellen der

Weiche weiterfahren kann.
Zu den Sensoren zählen sowohl Koppelspulen und Ortsbakensender als auch die Abtastung der Weichenlage durch induktive Näherungsschalter in der Bahn. Zukünftig soll auch durch magnetische Fahrsperren eine Richtungserkennung möglich sein. Eine Ausfallerkennung findet nur durch den Fahrer statt, und nur dann, wenn etwas nicht funktioniert, was wiederum an die Zentrale weitergeleitet wird.
Auch mit den Beschreibungsformalismen wird ähnlich manuell verfahren. Das Gleisnetz ist durch Hand- und CAD-Zeichnungen mit Eintragung der Koppelspulen und Weichen beschrieben, inzwischen wird mit einem Gleismesswagen eine GPS-gestützte Abbildung im Stadtplan erstellt, um Gleiszustände anzeigen zu können. Vernetzt werden sollen die einzelnen Gleispläne erst mit Umsetzung des neuen Systems. Die Sicherheitsbedingungen sind in der Dienstanweisung festgehalten, worin die Mitarbeiter viermal im Jahr geschult werden. Es gibt also keine booleschen Regeln, ob ein Fahrzeug in eine Route einfahren darf oder nicht, da ja der Fahrer alles manuell steuern kann. Über den Betrieb der BSAG wacht die Technische Aufsichtsbehörde, die alle Neueinführungen begutachten muß.

Vortrag von Herrn Hatesaul über das Steuerungssystem IMU100
Anhand von Folien des Herstellers Siemens wurden die technischen Details des IMU100 erläutert, von denen aber einige, wie die Funkfrequenzen und Abmessungen der Bauteile, nicht von Interesse sind. Es ist ein Bordinformationssystem, in dem alle möglichen Routen, auch Umleitungsstrecken, gespeichert sind. Die externe Kommunikation wird über Antenne, die interne über den seit 1980 als unverbindlichen VDV-Standard definierten Ibis Wagenbus geregelt. Die Fahrzeugortung läuft über Erkennung der gefahrenen Meter und wird auch dazu verwandt, Haltestellen anzusagen, was bisher der Fahrer per Knopfdruck erledigen mußte. Die Zeit wird per Funkuhr ermittelt. Die Weichen geben dem System keine Rückmeldung, das Fahrzeug wird nur über den Achskurzschluß und sein Gewicht registriert, so dass nachfolgende Züge die Weiche nicht stellen können. Mit dem IMU100-Empfänger können 16 frei definierbare Relais gesteuert werden. Das integrierte, aus finanziellen Erwägungen gleich mitgekaufte System ORBAS wird noch nicht eingesetzt. Diese ORtsBAkenSender verfügen über jeweils acht Relais.

Vortrag von Herrn Lübbers zum RBL (rechnergesteuertes Betriebsleitsystem) Die Ziele des RBL-Einsatzes sind klar umrissen: Durch die Verbesserung der Pünktlichkeit soll eine schnellere Beförderung errreicht werden, was für den Kunden eine Vereinfachung der Nutzung des Öffentlichen PersonenNahVerkehrs (ÖPNV), für den Mitarbeiter eine Erleichterung am Arbeitsplatz und für den Betrieb eine Kostenreduzierung durch mehr Effizienz bedeutet. Alle Fahrzeuge liefern viermal pro Minute ihre aktuelle Position, anhand der überprüft wird, ob der Fahrplan eingehalten wird. Die Daten werden über die digitalen Anzeigetafeln an bislang noch nicht allen Straßenbahnhaltestellen an die Kunden weitergeleitet: Sie wissen zumindest, wieviele Minuten die

Bahn von ihrer jetzigen Position aus normalerweise braucht. Außerdem werden Fahrzeuge angewiesen, auf andere zu warten, sobald nur noch alle zwanzig Minuten gefahren wird (Anschlusssicherung). Die Daten werden auch statistisch ausgewertet, um längerfristige Entwicklungen zu beobachten. Einen direkten Eingriff soll die Videoüberwachung an zentralen Haltestellen ermöglichen. Hohe Auslastung wird durch Einsatzwagen aufgefangen. Bereits genannt wurde die Automatisierung im Fahrzeug selber, der Fahrer muß nicht mehr die Haltestellenanzeige, Entwertersteuerung, Außenbeschilderung und Lichtsignalansteuerung übernehmen. Auch das Funksystem läuft über das RBL. Es gibt Einzelruf, Gruppenruf, Überfallruf und direkten Sprechfunk per Funkgerät, für den Datenfunk ist ein Zeitschlitz von 50ms vorgesehen.

2.4.2.3 Reflexion

Das Ergebnis des Besuches war zwiespältig. Fest stand, dass die Informationen nicht für alle Gruppen ausreichend waren, teilweise sogar dem Ansatz widersprachen. Da die BSAG aber nicht der einzige Anbieter ist, wurde der bisherige Ansatz weiterverfolgt und nach geeigneter Hardware gesucht. Hier wurden uns von der BSAG die Firmen Siemens, BBR und Hanning & Kahl empfohlen. Trotzdem hat es sich gelohnt, einen Einblick in die Praxis zu bekommen. Diese ist bislang aber noch sehr auf den Fahrer abgestimmt. Er hat die volle Kontrolle über die Bahn, damit aber auch die volle Verantwortung. Kein System verhindert, das ein Fahrer nach links abbiegt und in eine entgegenkommende Bahn fährt. Da er wie ein Kfz-Fahrer der StVO unterliegt und auch mit anderen Verkehrsteilnehmern interagieren muß, kann es keine vollkommen automatisierte Softwaresteuerung geben, was auch nie unser Ziel war. Den Fahrer von einem Teil seiner Arbeit zu entlasten und damit konzentrationsfähiger für andere Verkehrsteilnehmer zu machen, war unsere Auslegung, die der BSAG wohl eher, dass der Fahrer verantwortungsloser oder weniger konzentriert handelt. So erschien die Kontaktaufnahme mit der Berliner Straßenbahn sinnvoll, da die uns damals vorliegenden Informationen von Prof. Dr. Jan Peleska von dort stammten und diese Firma wohl auch schon weiter mit der Automatisierung war. Die Deutsche Bahn AG zu besuchen, machte aufgrund der unterschiedlichen Verfahrensweisen keinen Sinn.

2.4.3 Hanning & Kahl

2.4.3.1 Überblick

Während der Suche nach Informationen über Hardware-Elemente, die im Bereich Straßenbahnsteuerung genutzt werden, ist ein Mitglied der SI-Gruppe auf die Homepage von H&K gestoßen. Dies hat einen E-Mail Verkehr ausgelöst, durch den uns wichtigen Informationen zugeschickt wurden und von unserer Seite hat die Firma H&K Auskunft über unser Projekt bekommen.

Nach einiger Zeit hat sich herausgestellt, dass unser Projekt von Interesse für die Leute von H&K ist, so dass sie Kontakt mit unserem Lehrstuhl aufnehmen wollten. Das ist auch passiert. Als Ergebnis hat sich herausgestellt, dass H&K Interesse an unserem Projekt hat und gerne zum Besuch kommen wollte. Ein Termin wurde vereinbart.

2.4.3.2 Besuch von H&K

Am 9. Februar 2005 kam ein Treffen zwischen dem TRACS-Projekt und H&K zustande.

Das Treffen war in drei Teile gegliedert:

- Vortrag der AGBS
- Vortrag von H&K
- Vortrag von TRACS

2.4.3.2.1 Vortrag der AGBS Prof. Dr. Jan Peleska, als Leiter der AGBS, stellt die Aufgaben der Forschungsgruppe Betriebsysteme / Verteilte Systeme vor.
(Die entsprechenden Folien befinden sich im Verzeichnis docs/lectures/h+k und auf der TRACS Webseite)

2.4.3.2.2 Vortrag von H&K Rüdiger Mesterheide, Mitarbeiter von H&K präsentierte uns das Unternehmen H&K.

H&K wurde im Jahr 1898 in Oerlinghausen gegründet und seit dieser Zeit beschäftigt sich das Unternehmen mit Stellvorrichtungen und Steuerungen im Nahverkehrsbereich. Das Unternehmen ist in drei Geschäftsbereiche gegliedert: Bremsen, Dienstleistungen und Nahverkehr. Der Primäreinsatz liegt im Ausland.

Im Jahr 2004 hatte das Unternehmen 300 Mitarbeiter und erreichte einen Umsatz von 40 Mio. Euro.

Folien dazu befinden sich im Schrank des Projektraums unseres Projektes.

2.4.3.2.3 Vortrag von TRACS Verschiedene Projektteilnehmer haben einen Vortrag über unser Projekt gehalten, wobei die Projektaufgaben und der aktuelle Stand dargelegt wurden.

Folien dazu befinden sich auf der TRACS Webseite.

Ein Protokoll, welches das gesamte Treffen detailliert beschreibt, befindet sich im Verzeichnis docs/protocol sowie auf der TRACS Webseite.

2.4.3.3 Reflexion

Wir können das Treffen auf jeden Fall als für uns gelungen beurteilen. Schade war nur, dass die Kontaktaufnahme zu so einem späten Zeitpunkt (kurz vor dem Projektende) stattfand. Aus Zeitgründen konnten die verbleibenden Fragen nicht mehr diskutiert werden. Diese wurden per Mail an H&K, stellvertretend Herr Mesterheide, zugeschickt.

H&K hat sich dazu bereit erklärt, unser Weltmodell durchzulesen und uns entsprechende Kommentare zuzuschicken.

Während des inoffizielle Teils des Treffens hat sich herausgestellt, dass H&K nach unserem Projektabschluss weiter mit der AGBS arbeiten möchte. Es ist auch möglich, dass Diplomarbeitsthemen gestellt werden, an denen H&K Interesse hätte und Projektteilnehmer hätten dadurch die Möglichkeit, ihre Diplomarbeiten auf Basis der vorgeschlagenen Themen zu schreiben.

Marcin Dysarz

Kapitel 3

Problembeschreibung

Die Problemstellung besteht darin, Straßenbahnen sicher duch ein Gleisnetz zu leiten. Dabei soll auf möglichst geringe Kosten geachtet werden, weshalb möglichst viele Schritte bei der Anpassung auf ein neues Gleisnetz automatisiert werden sollen.
Beim Durchleiten einer Straßenbahn muss sichergestellt werden, dass es nicht zu Kollisionen zwischen Straßenbahnen kommen kann.

3.1 Gleisnetzsteuerung

Um das Gleisnetz zu steuern, müssen Weichen und Signale in die richtige Position gesschaltet werden, so dass Kollisionen nicht geschehen. Da Straßenbahnen auf Sicht fahren, muss die Gleisnetzsteuerung nicht dafür sorgen, dass es keine Auffahrunfälle gibt. Die benötigten Informationen über die Positionen und gewünschten Routen der Bahnen kann das System durch Sensoren erhalten. Dabei muss beachtet werden, dass es verschiedene Typen von Sensoren, Weichen und Signalen gibt, deren Eigenschaften zusammen mit den Einflüssen der Umwelt im Abschnitt 2 auf Seite 10 beschrieben werden.

3.2 Gleisnetzsteuerungssysteme

Gleisnetzsteuerungssysteme müssen Bahnen sicher durch das System auf den gewünschten Routen durchleiten. Dazu müssen die Routen unterschieden werden, es muss also erkannt werden, auf welcher Route die Bahnen das System durchfahren soll. Die Gleisnetzelemente müssen dann so geschaltet werden, dass das Durchfahren der Routen möglich wird.
Dabei muss auf Sicherheit geachtet werden. Das bedeutet, dass es durch die Gleisnetzsteuerung nicht zu Sach- und Personenschäden kommen darf. Dazu müssen Konflikte erkannt werden, Angeforderte Routen dürfen also nur dann freigeschaltet werden, wenn

sie nicht im Konflikt zueinander oder zu bereits freigeschalteten Routen stehen. Es darf also nicht zu Kollisionen zwischen Bahnen kommen. Dazu gehört auch, dass erkannt wird, wenn Routen wieder frei sind, was durch Auswertung der Sensorinformationen erfolgen muss, damit dazu im Konflikt stehende Routen dann auch freigeschaltet werden können.
Weiterhin ist zu sicherzustellen, dass durch geeignete Signalstellungen verhindert wird, dass Bahnen nicht auf Weichen fahren, die gerade geschaltet werden und es dadurch zu Entgleisungen kommen kann.
Außerdem ist zu gewährleisten, dass eine Bahn nicht unnötig lange wartet, es muss also Fairness gewährleistet werden. Es dürfen also nicht ständig dieselben Routen freigeschaltet werden, wenn auch dazu im Konflikt stehende Routen angefordert sind.
In den folgenden beiden Abschnitten wird dann zwischen automatischer und manueller Generierung verglichen.

3.3 Deren Generierung

In der manuellen Entwicklung gibt es neben dem wiederkehrenden Entwicklungsaufwand das Problem, dass mehrere Schritte der Entwicklung relativ aufwendig von Hand auf ihre Sicherheit geprüft werden müssen, was eine Menge Zeit kostet und damit auch Geld, weil diese Schritte für jedes Gleisnetz erneut durchgeführt werden müssen.
Es muss für jedes Gleisnetz von Hand die gesamte Entwicklungsarbeit für das Steuerungssystem erneut ausgeführt werden. Daneben gibt es das Problem, dass auch die Validierung für jedes Gleisnetz neu von Hand durch geführt werden. Die händische Programmierung ist außerdem noch fehlerträchtig und führt zu Inkonsistenzen.

3.4 Deren automatische Generierung

Wenn die Gleisnetzsteuerungssysteme hingegen automatisch generiert werden könnten, würden die obengenannten Probleme reduziert werden. Dafür ergibt sich dann das zusätzliche Problem, dass die automatische Generierung auch für jedes Gleisnetz funktionieren muss, welches dem im Abschnitt 2 auf Seite 10 beschriebenen Weltmodell entspricht. Aber mit genau dieser Universalität kann dann vermieden werden, dass für jedes Gleisnetz Code von Hand geschrieben werden muss, so dass der Code dafür wiederverwendbar ist. Durch die Wiederverwendbarkeit muss nicht mehr alles neu entwickelt werden, wodurch der Entwicklungsaufwand verkleinert wird. Auch der Verifikationsaufwand kann durch die Automatisierung minimiert werden, auch dadurch dass wiederverwendbare Bestandteile nur einmal verifiziert werden müssen. Im Wesentlichen müssen nur die für das jeweilige Gleisnetz spezifischen Daten jeweils eingegeben werden.
Ziel von TRACS ist es genau so eine automatische Generierung zu entwerfen. Der dazu

zu entwerfende Entwicklungs- und Verifikationsprozess, also wie das Steuersystem automatisch generiert und verifiziert wird, wird in den nächten beiden Kapiteln beschrieben.

Andreas Kemnade

Kapitel 4

Der Lösungsansatz

4.1 Architektur

In diesem Abschnitt wird die Architektur des von TRACS entwickelten Gleisnetzsteuerungssystems beschrieben.
Die Abbildung 4.1 auf der nächsten Seite zeigt den heute gebräuchlichen Ansatz zur Entwicklung von Gleisnetzsteuerungssystemen. Wie schon in der Einleitung erwähnt (Abschnitt 1.4 auf Seite 7) wird das Steuerungssystem weitgehend manuell anhand der vorliegenden Gleisnetzinformationen erstellt. Dadurch ist für jeden Controller ein eigener Entwicklungsprozess notwendig.
Das Projekt TRACS verfolgt einen neuen Ansatz. Die Abbildung 4.2 auf Seite 40 zeigt die Architektur dieses Ansatzes. In der Abbildung werden alle beteiligten Komponenten und die Beziehungen zwischen den einzelnen Komponenten dargestellt.

- CAD-Daten

  Es wird eine graphische Darstellung des betreffenden Gleisnetzes am Computer erstellt. Dies sind die sogenannten CAD-Daten.

- Verschlusstabellen

  Anhand des gegebenen Gleisnetzes werden die Verschlusstabellen erstellt. In den Verschlusstabellen werden Routeninformationen für das jeweilige Gleisnetz definiert.

- Domänenspezifische Beschreibungssprache (DSL)

  Es wird eine DSL (Domain Specific Language) entwickelt. Eine DSL ist eine Sprache, die die Begriffe und Konzepte ihres Anwendungsgebietes verwendet. Die entwickelte DSL wird als TND (Tram Network Description) bezeichnet. Sie enthält eine Beschreibung des Gleisnetzes, eine Beschreibung der geplanten Routen sowie eine Beschreibung der verwendeten Hardware.

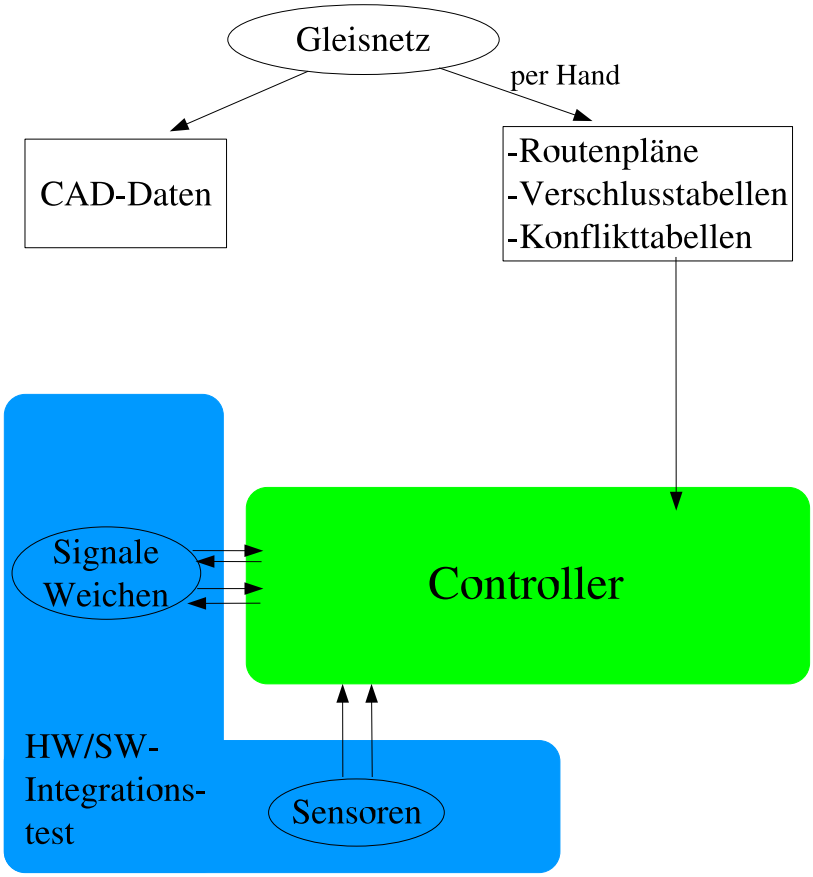


Abbildung 4.1: Herkömmlicher Ansatz bei der Entwicklung von Gleisnetzsteuerungssystemen

- Compiler

  Ein Compiler erzeugt aus der TND binäre Projektierungsdaten, welche vom Steuerinterpreter effizient verarbeitet werden können.

- Steuerungssoftware

  „Die eigentliche Steuerungsaufgabe übernimmt eine Softwarekomponente, die interpretativ auf Grundlage der Projektierungsdaten arbeitet.“ [TRA]

- Verifikation

  Die Sicherheit der Projektierungsdaten wird mit Hilfe von Model Checking überprüft.

- Test

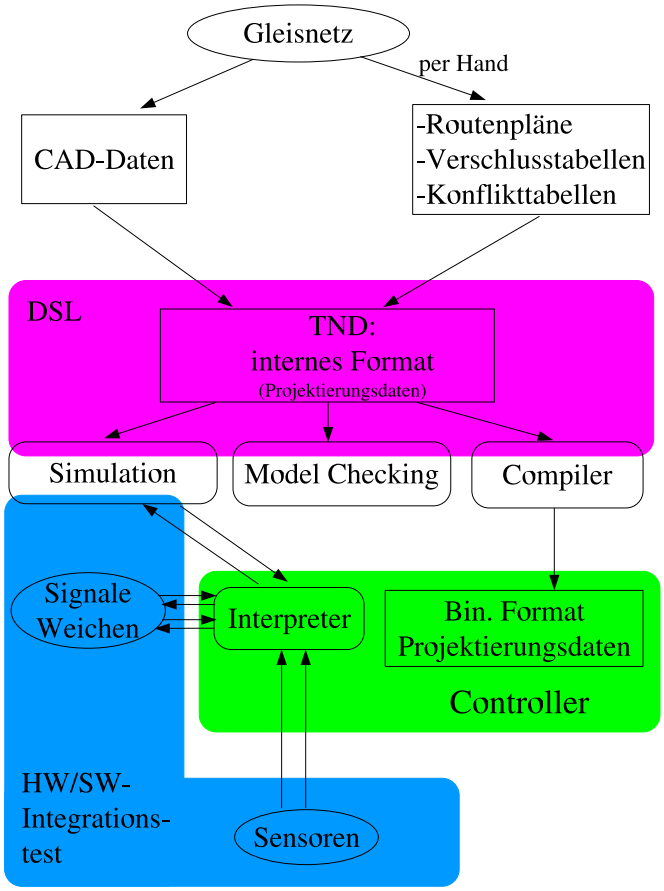


Abbildung 4.2: Neuer Ansatz des Projektes

Das integrierte Hardware/Software-System wird mittels einer automatisierten Test Suite in Bezug auf sein korrektes Verhalten geprüft.

- Simulation

  Durch einen Simulator kann die Steuerung des jeweiligen Gleisnetzes am Computer simuliert werden.

Taffou Happi

4.2 Automatische Generierung

Eines der Hauptziele des Projekts TRACS ist die Minimierung des Aufwandes, um ein Steuerungssytem für ein neues Gleisnetz zu erstellen. Aus diesem Grund ist das von uns entwickelte System primär auf Wiederverwendbarkeit ausgelegt, so dass es für

möglichst viele Gleisnetze verwendet werden kann. Dazu wird für jedes Gleisnetz in mehreren Schritten aus einer graphischen Beschreibung ein fertiges Steuerungssystem generiert.

4.2.1 Wiederverwendbare Komponenten

Ein Grossteil der Automatisierung wird dadurch erreicht, dass die meisten in TRACS verwendeten Komponenten wiederverwendbar sind, also eine generische Lösung für die jeweilige Aufgabe darstellen, die erst später mit gleisnetzspezifischen Daten ergänzt werden.

4.2.1.1 Netzgraph / Konvertierungstools

Der erste Schritt besteht in einer Auswertung der graphischen Beschreibung durch die von uns Netzgraph genannte Softwarekomponente, welche die vom Anwender erstellte DXF-Datei einliest, analysiert und schliesslich eine für unseren Compiler verständliche TND Beschreibung übersetzt. Diese Übersetzung umfasst u.a. das Erkennen von Gleisnetzelementen, deren Eigenschaften und Beziehungen, da diese die Grundlage unserer TND darstellen.
Da nicht alle für eine vollständige TND benötigten Informationen wie z.B. Routeninformationen in einer graphischen Beschreibung sinnvoll aufgehoben sind wird diese "Roh"-TND noch mit einem zusätzlichen Tool namens TND-Builder nachbearbeitet, um diese Informationen einzufügen. Das Zwischenprodukt am Ende dieses Schrittes ist eine TND, die sämtliche für das Steuerungssystem und dessen Verifikation notwendigen Daten enthält.

Eine genaue Beschreibung findet sich in den Abschnitten 7.2 auf Seite 82 und 7.3 auf Seite 123.

4.2.1.2 Compiler

Die im ersten Schritt generierte TND stellt nicht nur den Ausgangspunkt für die spätere Verifikation dar, sondern auch für den weiteren Generierungsprozess durch unseren TND-Compiler. Dieser transformiert die in Textform vorliegende TND in ein binäres Format, welches später die konkreten Anweisungen für das spezielle Steuerungssystem liefert. Hierbei wird die TND auf Korrektheit geprüft, aus den vom Anwender definierten Verschlusstabellen werden konkrete Sicherheitsbedingungen generiert und aus den Routendefinitionen werden konkrete Schaltinformationen extrahiert. Ausserdem werden die notwendigen Daten gewonnen, damit der Steuerinterpreter später auch die richtigen Treiber für die im Gleisnetz verwendete Hardware benutzt.

Eine genaue Beschreibung findet sich in Abschnitt 7.4 auf Seite 135.

4.2.1.3 Steuerinterpreter

Während die bisherigen Schritte nur Informationen ausgewertet und übersetzt haben ist der Steuerinterpreter zuständig für die Ausführung der generierten Anweisungen. Es handelt sich hierbei um eine generische Steuerungssoftware, die nur im Zusammenspiel mit den vom Compiler erzeugten binären Projektierungsdaten sinnvoll benutzbar ist, da diese die konkreten Steuerungsinformationen enthalten

Eine genaue Beschreibung findet sich in Abschnitt 7.5 auf Seite 171.

4.2.1.4 Treiber

Die letzte Komponente sind schliesslich die einzelnen Hardwaretreiber, die am Ende des Generierungsprozesses ins Steuerungssystem eingebunden werden. Sie sind das entscheidende Glied in der Kommunikation zwischen Steuerinterpreter und den verschiedenen Hardwarelementen im Gleisnetz und dafür verantwortlich, dass die vom Steuerinterpreter gegebenen Anweisungen korrekt umgesetzt werden.

4.2.2 Generierte Anteile

Zwar sind die wiederverwendbaren Komponenten der Schlüssel von TRACS, allerdings werden im Verlauf des Generierungsprozesses auch diverse gleisnetzspezifische Komponenten automatisch generiert, die die konkreten Daten des jeweiligen Gleisnetzes wiederspiegeln. Nur mit diesen können die generischen Komponenten schliesslich sinnvoll eingesetzt werden.

4.2.2.1 TND-Zwischendarstellung

Unsere Tram Network Description enthält den kompletten Aufbau eines Gleisnetzes, d.h. die Topologie, sämtliche Routendaten und Angaben zu Gleisnetzelementen, kurz gesagt alles was für ein Steuerungssystem benötigt wird. Die TND ist damit die massgebende Datenquelle für unseren Generierungs- und Verifikationsprozess.

Eine genaue Beschreibung findet sich in Abschnitt 7.1 auf Seite 59.

4.2.2.2 binäre Projektierungsdaten

Während die TND eine menschenlesbare Datei mit mehreren Verwendungszwecken ist benötigt das Steuerungssystem eine möglichst einfach zu verarbeitende und hocheffizien-

te Form von Steueranweisungen. Diese werden in Form der binären Projektierungsdaten generiert, die genau angeben welche Hardwareelemente vorhanden sind sowie wann und wie sie zu schalten sind.

Marius Mauch

4.3 Verifikation und automatisierter Test

4.3.1 Model Checking

Model Checking ist ein Verfahren, um zu überprüfen, ob in einem System bestimmte Eigenschaften gelten. Dazu muss ein Modell des Systems erstellt werden und es müssen die zu überprüfenden Eigenschaften spezifiziert werden. Hier soll Model Checking eingesetzt werden, um die Sicherheitseigenschaften zu überprüfen.
Eine wesentliche Aufgabe des Steuerungssystems ist es, das Auftreten von Entgleisungen und Kollisionen zu verhindern. Anhand der Beschreibung des jeweiligen Gleisnetzes lässt sich ein Modell des Gleisnetzes erstellen und es lassen sich die Sicherheitsbedingungen bestimmen, die in dem Gleisnetz gelten müssen. Das Steuerungssystem muss die Signale und Weichen so schalten, dass keine dieser Sicherheitsbedingungen verletzt wird.
In welcher Situation die Signale und Weichen für das jeweilige Gleisnetz wie geschaltet werden sollen, wird in den Projektierungsdaten definiert. Mit Model Checking wird überprüft, ob die in den Projektierungsdaten definierten Anforderungen zu keinem Zustand führen können, in dem eine Sicherheitsbedingung nicht erfüllt ist.
Es wird ein formales Modell erstellt. Dazu werden die Spezifikation des Systems, die in der TND-Netzwerkbeschreibung enthaltenen Informationen sowie die in den Projektierungsdaten enthaltenen Informationen verwendet.
Model Checking wird ausführlich im Abschnitt 7.6 auf Seite 203 behandelt.

Ruben Rothaupt

4.3.2 Automatisierter Test

Zur Überprüfung des gesamten Hardware/Software Systems wird ein automatisierter Vorgang benutzt. Die Testautomatisierung ermöglicht eine schnellere und sicherere Testausführung als würde es ein Mensch machen. In einer kürzeren Zeit kann man eine größere Testabdeckung und dadurch eine Verringerung der Qualitätsrisiken erreichen. Es stellt sich heraus, dass bei der wiederholten Testdurchführung, gegenüber manuellen Tests, die Testprozeduren jedes Mal reproduziert werden können. Die Vorbereitung der Testprozeduren, im Vergleich zur manuellen Testausführung, fordert mehr Initialaufwand. Der Durchführungaufwand ist aber gering.
Die Schnittstellen-Definition spielt bei den automatisierten Tests eine wichtige Rolle. Sie dienen als Kommunikationskanäle zwischen der Testumgebung und dem Testling (zur

Simulation und Erfassung der Reaktionen des zu testenden Systems).
In dem Testvorgang werden keine konkreten Testdaten fest gespeichert. Die werden erst nach dem Einlesen der dem Test zugehörigen Projektierungsdaten, während der Ausführung, generiert. Durch dieses Schema ist Wiederverwendbarkeit einer Testprozedur für mehrere Testkonfigurationen garantiert.

Als Testwerkzeug benutzen wir die Software "RT-Tester", der alle oben genannten Funktionalitäten uns anbietet. Die genaue Beschreibung ist im Kapitel 7.7.3.1 auf Seite 259 dargestellt.

Die vollständige Beschreibung der Testprozeduren findet man in den Abschnitten 7.7.2 auf Seite 255, 7.7.4 auf Seite 262, 7.7.5 auf Seite 267.

Marcin Dysarz

Kapitel 5

Entwicklungsprozess

In diesem Kapitel wird der Prozess beschrieben, wie in TRACS ein Steuersystem generiert wird.

5.1 Überblick

Um einen Verifikationsprozess zu ermöglichen, muss auch der Entwicklungsprozess systematisch beschrieben werden, damit klar zu sehen ist, welcher Entwicklungsschritt in welcher Form verifiziert werden kann. Im Abschnitt 5.2 werden die Grenzen des TRACS-Systems dargestellt. Im Rahmen des Entwicklungsprozesses wird auf von uns entwickelte Komponenten zurückgegriffen, die benötigt werden, um eine ausführbare Steuerungssoftware zu erzeugen. Der Abschnitt 5.3 auf der nächsten Seite gibt einen Überblick über die dazu von uns entwickelten Komponenten. Der Erstellungsprozess, der die einzelnen Schritte von den Systemeingaben bis zum Endprodukt beschreibt, wird in Abschnitt 5.4 auf Seite 47 erläutert.

Taffou Happi, Deng Zhou, Ruben Rothaupt, überarbeitet von Andreas Kemnade

5.2 Umfang des von TRACS gelieferten Systems

Die Aufgabe von TRACS ist es, eine ausführbare Steuerungssoftware für ein konkretes Gleisnetz des Straßenbahnbetreibers zu erzeugen. Dazu werden bestimmte Eingaben benötigt. Man kann dabei zwischen Eingaben unterscheiden, die für jedes Gleisnetz neu gemacht werden müssen und Eingaben, die für jedes Gleisnetz konstant sind. Für jedes konkrete Gleisnetz muss eine Gleisnetzbeschreibung vorliegen, die zu verwendende Hardware muss vom Kunden spezifiziert werden und die Verschlusstabellen müssen erstellt werden. Die Verschlusstabellen enthalten Angaben über zu befahrende Routen, Routenkonflikte und die für die jeweilige Route benötigten Weichen- und Signalstellungen. Die Systemanforderungen und die PC-Hardware sind für jedes Gleisnetz gleich.

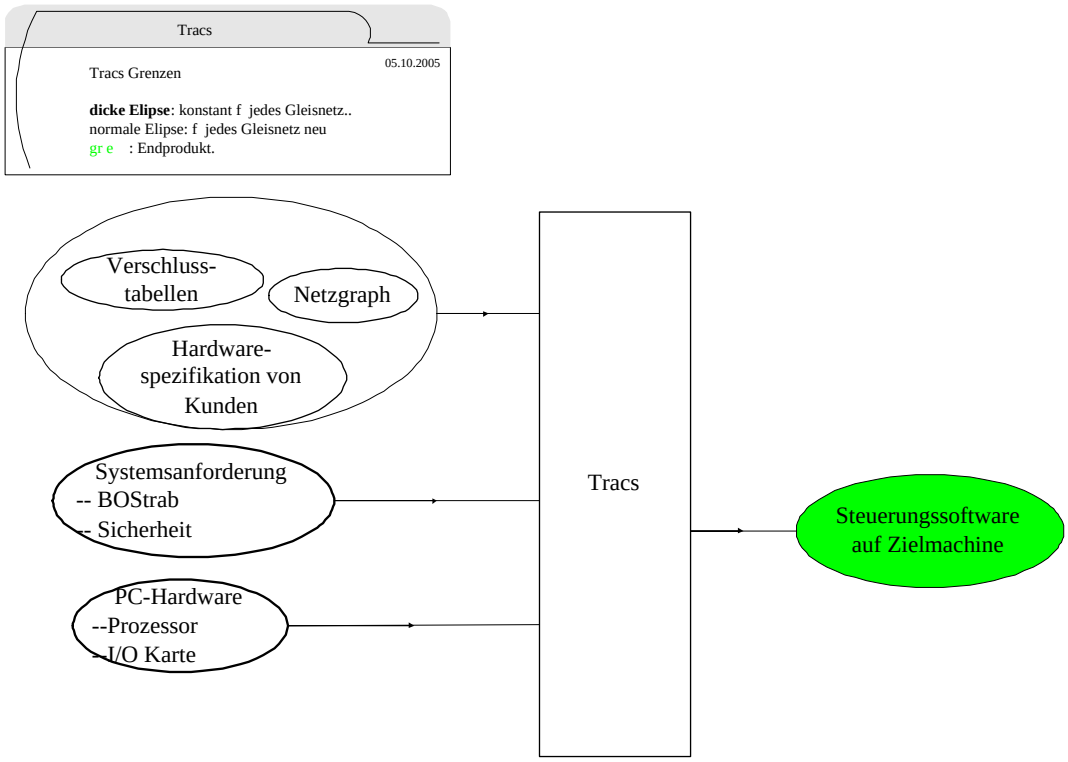


Abbildung 5.1: Umfang des von con TRACS gelieferten Systems

Taffou Happi, Deng Zhou, Ruben Rothaupt, überarbeitet von Andreas Kemnade

5.3 Entwicklungsschritte

Anhand der Abbildung 5.2 auf der nächsten Seite wird der Ansatz zur Entwicklung der wiederverwendbaren TRACS-Komponenten aufgezeigt.
Die folgenden Komponenten wurden im Rahmen des Entwicklungsprozesses entwickelt:

- CAD-TND-Konverter
- Komponente zur Transformation der Verschlusstabellen in die TND
- Komponente zur Transformation der Hardwarebeschreibung in die TND
- TND-Compiler
- Steuerinterpreter
- Treiberbibliothek

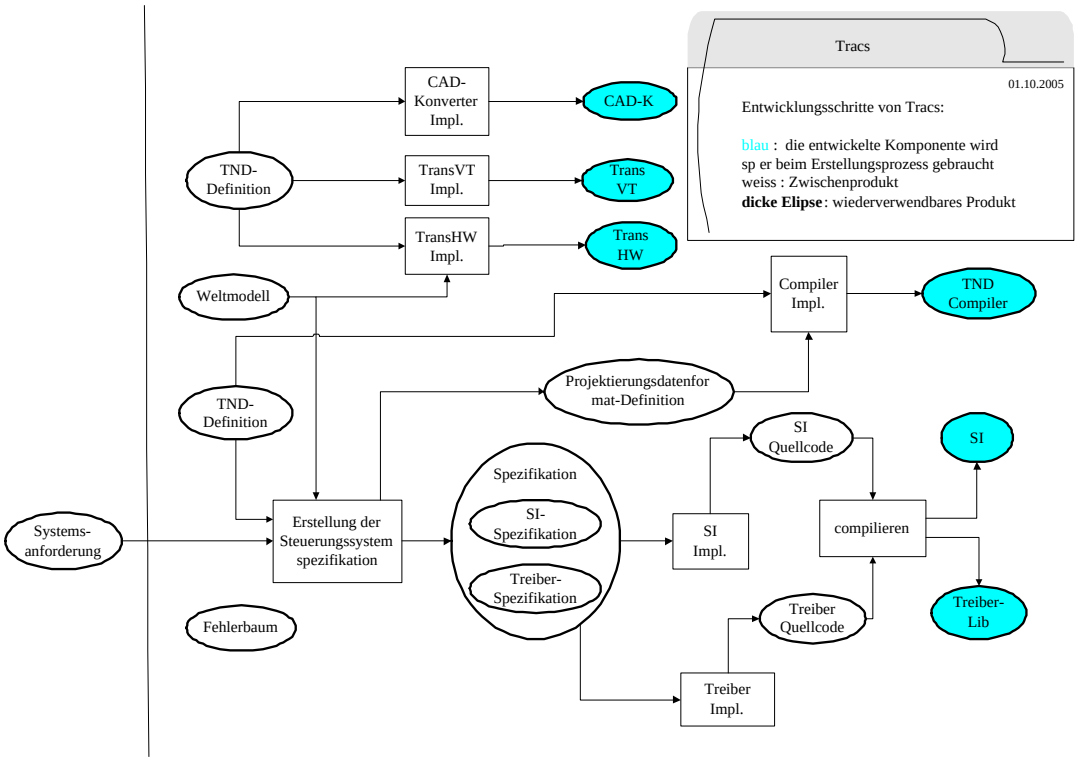


Abbildung 5.2: Entwicklungsprozess

Diese Komponenten werden für den Erstellungsprozess benötigt. Der Steuerinterpreter, die Treiber und der TND-Compiler werden auf Grundlage einer Spezifikation des Steuerungssystems implemetiert. Diese Systemspezifikation besteht aus SI-Spezifikation, Treiber-Spezifikation und Definition des Projektierungsdatenformats. Zur Erstellung der Systemspezifikation werden die Systemanforderungen, das Weltmodell und die TND-Definitionen benötigt.

Taffou Happi, Deng Zhou, Ruben Rothaupt, überarbeitet von Andreas Kemnade

5.4 Erstellungsprozess

Die Notwendigkeit eines klar definierten Erstellungsprozesses ergibt sich aus dem Entwicklungprozess, denn nur dadurch können auch die Verifikationsmaßnahmen automatisch durchgeführt werden. Anhand der Abbildung 5.3 auf der nächsten Seite wird der Ansatz zur Erstellung einer kundengleisnetzspezifischen Steuerungssoftware, die auf einer Zielmaschine läuft, aufgezeigt.
Der gesamte Erstellungsprozess besteht aus folgenden Schritten:

- Übertragung der jeweilligen Gleiznetzdarstellung in die TND

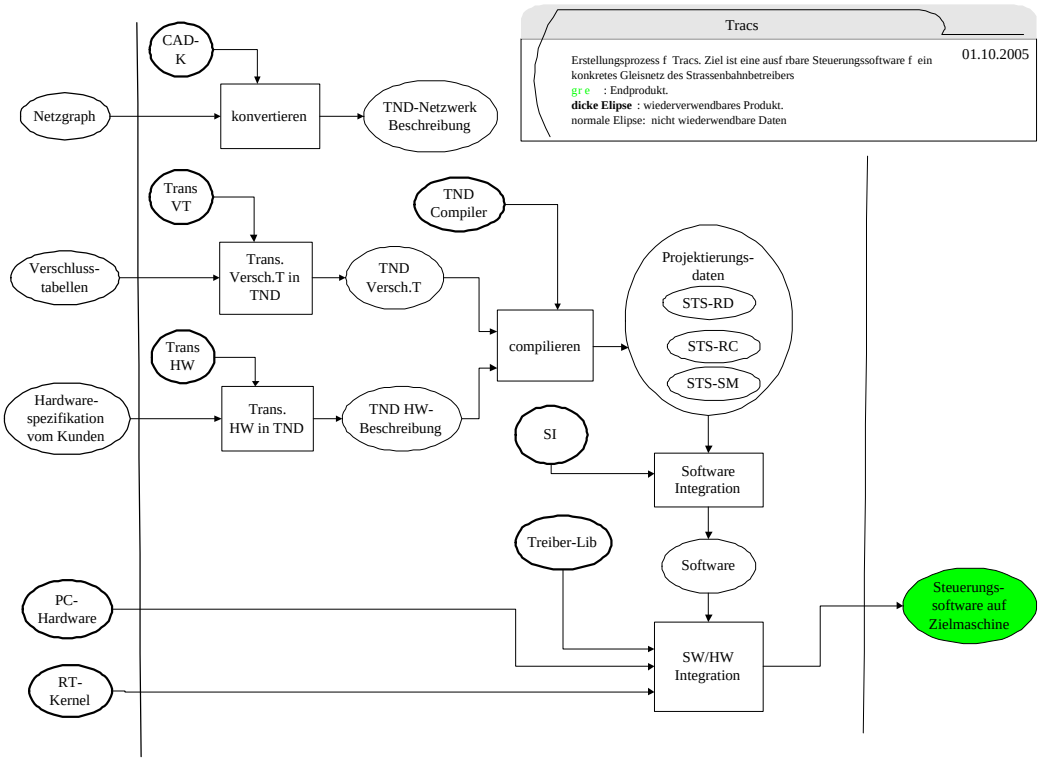


Abbildung 5.3: Erstellungsprozess

- Übertragung der vom Kunden gelieferten Hardwarespezifikation in die TND
- Übertragung der vom Kunden gelieferten Verschlusstabellen in die TND
- Erzeugen von Projektierungsdaten für das jeweillige Gleisnetz aus der TND
- Erzeugen der Steuerungssoftware aus Steuerinterpreter und konkreten Projektierungsdaten
- Erzeugen des Zielproduktes durch Integration der Steuerungssoftware in der Hardware

Die für jedes Gleisnetz benötigten Eingaben sind die Gleisnetzdarstellung im Netzgraph, die Verschlusstabelleneingaben sowie die Hardwarespezifikation vom Kunden. Diese Eingaben sollen in der TND repräsentiert werden. Die TND besteht also aus einer Netzwerkbeschreibung, einer Verschlusstabellendarstellung und einer Hardwarebeschreibung.
Ein Konverter erzeugt aus dem Netzgraphen die TND-Netzwerkbeschreibung und die TND-Hardwarebeschreibung. Die in den Verschlusstabellen gemachten Angaben werden in die TND-Verschlusstabellendarstellung übertragen. Es wird ein Compiler entwickelt,

welcher aus der TND- Verschlusstabellendarstellung und der TND-Hardwarebeschreibung die Projektierungsdaten für das jeweilige Gleisnetz erzeugt. Die Projektierungsdaten dienen dazu, um das Verhalten des Steuerinterpreters an das Gleisnetz anzupassen. Der wiederverwendbare Steuerinterpreter bildet daher mit den jeweiligen Projektierungsdaten zusammen die Software.
Um das Endprodukt, die Steuerungssoftware auf der Zielmaschine, zu erreichen, muss die Software noch in die verwendete Hardware integriert werden.

Taffou Happi, Deng Zhou, Ruben Rothaupt, überarbeitet von Andreas Kemnade

Kapitel 6

Validierungs-, Verifikations- und Testprozess

6.1 Überblick

Im Laufe des Projektes wurde immer wieder die Frage gestellt, wieso man davon ausgehen kann, dass es sicher ist in eine Straßenbahn einzusteigen. Hinter dieser Frage steht die Anforderung an die Sicherheit eines Steuerungssystems.
Wenn beispielsweise ein Signal oder eine Weiche falsch gestellt sind oder ein Signal eine Route freigibt, bevor zu der Route gehörige Weichen richtig gestellt sind, kann es zu Kollisionen oder Entgleisungen kommen.
Um zu garantieren, dass das Steuerungssystem das gewünschte Verhalten zeigt, muss es validiert, verifiziert und getestet werden, bevor es eingesetzt wird.
In diesem Abschnitt wird der Prozess beschrieben, durch den gewährleistet wird, dass das entwickelte Steuerungssystem sicher ist und es wird dargestellt, wie dieser Prozess automatisiert werden kann.
Das Prinzip der Validierungs-, Verifikations- und Testmethode sieht folgendermaßen aus:

- Validierung: stellt sicher, dass die Sicherheitszusagen und die Softwarespezifikation geeignet und vollständig sind
- Verifikation: stellt sicher, dass jedes einzelne Entwicklungsprodukt mit der Spezifikation konsistent ist (vollständig)
- Test: stellt sicher, dass das implementierte Steuerungssystem die erwartete Aufgabe erfüllt (unvollständig)

Eine kurze Zusammenfassung des Entwicklungsprozesses ist in Abbildung 6.1 auf der nächsten Seite zu sehen.
Die Abbildung zeigt, dass es drei wichtige Schritte in der Entwicklungsphase gibt:

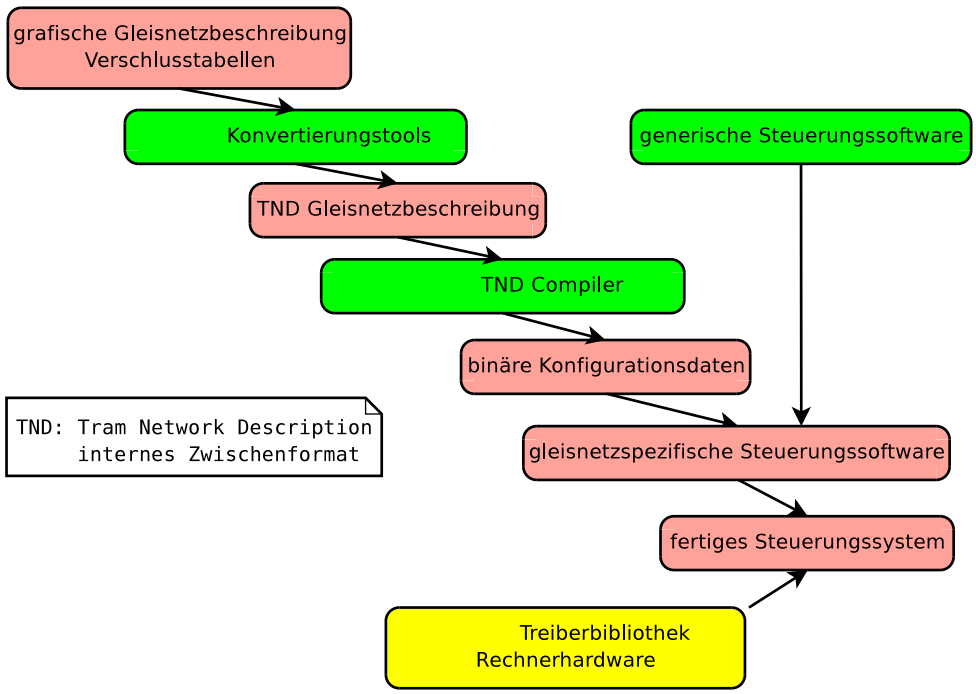


Abbildung 6.1: Entwicklungsprozess

- Erstellung einer domänenspezifischen Beschreibung (TND)
- Erstellung einer gleisnetzspezifischen Steuerungssoftware
- Erstellung des fertigen Steuerungssystems

Die TND besteht aus einer Beschreibung des jeweiligen Gleisnetzes, einer Hardwarebeschreibung und einer Routenbeschreibung, die die in den Verschlusstabellen gemachten Angaben wiedergibt. Bevor die TND vom Compiler verarbeitet wird, soll überprüft werden, ob sie auch alle für das Gleisnetz spezifizierten Hardwareelemente und die Nachbarschaftsbeziehungen zwischen diesen Elementen korrekt wiedergibt. Zudem soll überprüft werden, ob alle Routen gültig sind, also so befahren werden können, wie sie definiert wurden, und alle Routenkonflikte fehlerfrei definiert wurden.
Die wiederverwendbare Steuerungssoftware ist nicht vom jeweiligen zu steuernden Gleisnetz abhängig und wird durch ausführliche Tests und Review des Programmcodes (optional) überprüft.
Weiter wird überprüft, ob die vom Compiler erstellten, gleisnetzspezifischen Projektierungsdaten eine sichere Steuerung des Gleisnetzes ermöglichen. Dazu wird das System als Transitionssystem modelliert. In den Projektierungsdaten ist dabei definiert, wann die Weichen und Signale wie geschaltet werden sollen. Es wird vollständig überprüft, ob keine Transition dazu führen kann, dass eine Sicherheitsbedingung verletzt wird.

Nun erfolgt der letzte Schritt. Dabei wird das fertige Steuerungssystem, das aus wiederverwendbarer Steuerungssoftware, gleisnetzspezifischen Projektierungsdaten, Treibern und Hardware besteht, ausführlich getestet. Diese Tests sind erforderlich, um zu überprüfen, ob das System auch nach der Integration mit den verwendeten Treibern und der verwendeten Hardware noch das korrekte Verhalten zeigt.
Demzufolge müssen nun folgende Validierungs-, Verifikations- und Testschritte ausgeführt werden:

- Validierung einer TND-Datei
  - Die Validierung der Netzwerkbeschreibung wird im Abschnitt 6.2 beschrieben.
  - Die Validierung der Routenbeschreibung wird im Abschnitt 6.3 auf der nächsten Seite beschrieben.
  - Die Validierung der Hardwarebeschreibung wird im Abschnitt 6.4 auf Seite 54 beschrieben.
- Verifikation der Projektierungsdaten
  - Auf die Sicherheitsüberprüfung der Projektierungsdaten mit Model Checking wird im Abschnitt 6.5 auf Seite 55 eingegangen.
- Test
  - Im Abschnitt 6.6 auf Seite 56 werden die Tests der gleisnetzspezifischen Steuerungssoftware erläutert. Dies beinhaltet Modultests und Softwareintegrationstests.
  - Die Tests des fertigen Steuerungssystems werden im Abschnitt 6.8 auf Seite 58 erläutert. Dazu werden Hardware-in-the-Loop-Tests ausgeführt.

Wenn diese Schritte erfolgreich ausgeführt werden, ist nachgewiesen, dass das Steuerungssystem seine beabsichtigte Aufgabe erfüllt.

Deng Zhou, Ruben Rothaupt

6.2 Validierung der TND-Netzwerkbeschreibung

Die TND-Netzwerkbeschreibung wird durch einen Konverter aus dem Netzgraphen erzeugt.
Der Simulator verwendet einen Parser, um die Netzwerkbeschreibung einzulesen und eine Visualisierung des Gleisnetzes zu erstellen. Diese Visualisierung wird nun mit der

ursprünglichen Gleisnetzdarstellung im Netzgraphen verglichen. Wenn die beiden Darstellungen äquivalent sind, wurde gezeigt, dass der Konverter seine Funktion erfüllt hat und die Netzwerkbeschreibung innerhalb der TND auch das tatsächlich betrachtete Gleisnetz wiedergibt. Wie der Simulator eine Visualisierung des Gleisnetzes erstellt, wird im Abschnitt 7.8 auf Seite 313 beschrieben.

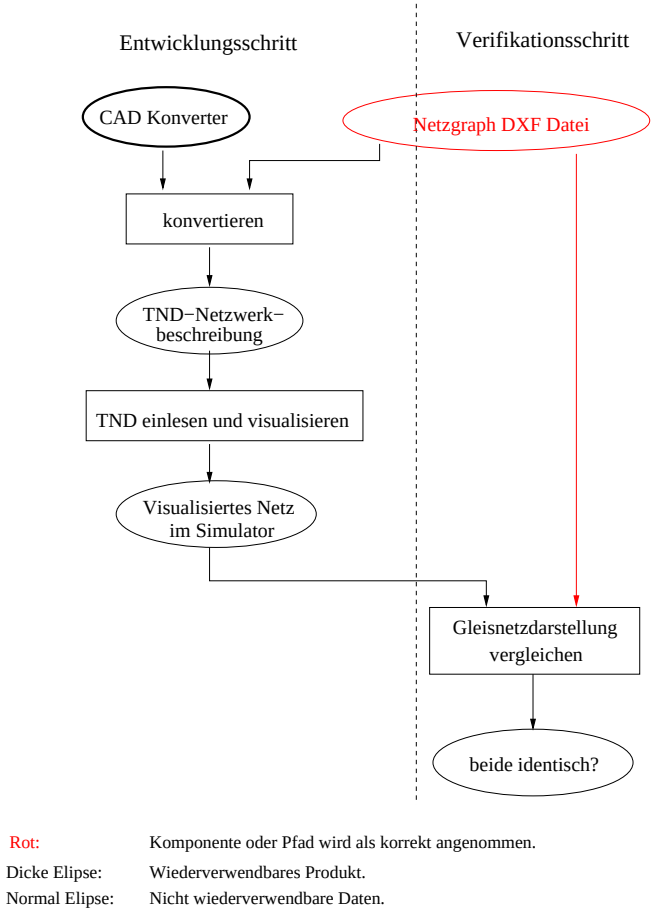


Abbildung 6.2: Validierung der TND-Netzwerkbeschreibung

Deng Zhou, Ruben Rothaupt

6.3 Validierung der TND-Routenbeschreibung

Die TND-Routenbeschreibung enthält die in den Verschlusstabellen gemachten Angaben.
Zur Überpüfung der Routenbeschreibung wird Model Checking verwendet.

Dazu wird aus der TND-Netzwerkbeschreibung ein Modell des Gleisnetzes abgeleitet. Nun kann überprüft werden, ob die Routen auch so befahrbar sind, wie sie definiert wurden und ob alle Routenkonflikte richtig definiert wurden. Auf diesen Vorgang wird im Abschnitt 7.6 auf Seite 203 näher eingegangen.

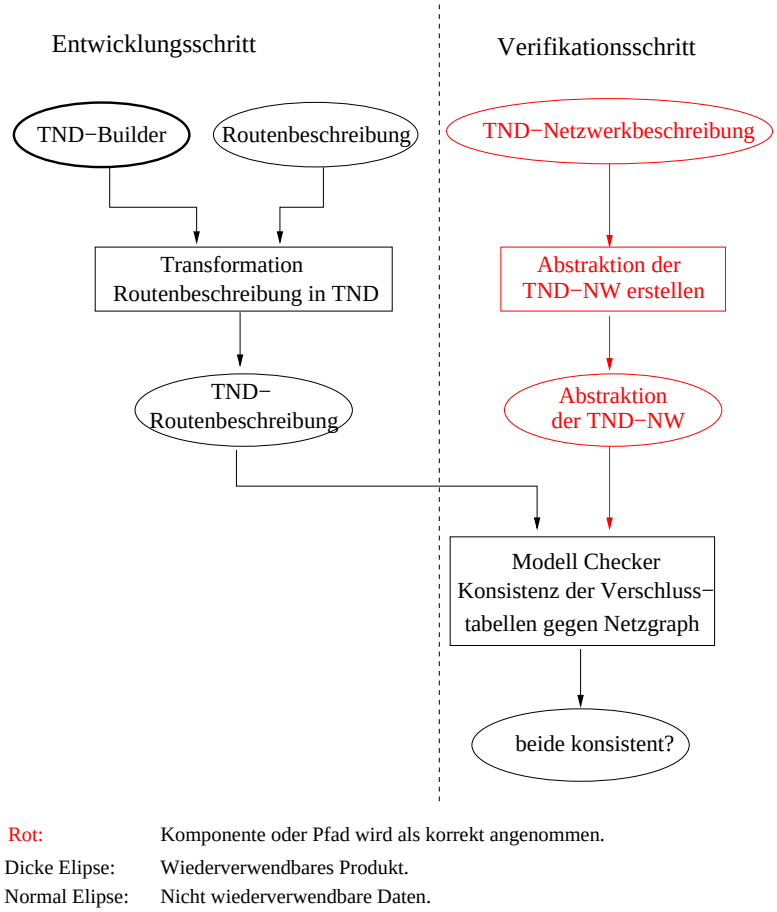


Abbildung 6.3: Validierung der TND-Routenbeschreibung

Deng Zhou, Ruben Rothaupt

6.4 Validierung der TND-Hardwarebeschreibung

Die TND-Hardwarebeschreibung enthält die Beschreibungen der vom Kunden erhalteten Hardware.
Dass die TND-Hardwarebeschreibung tatsächlich mit der vom Kunden gelieferten Hardwarebeschreibung übereinstimmt, wird durch einen manuellen Vergleich geprüft. Beide

Beschreibungen werden in Form einer Review-Methode durchgesehen, um sicherzustellen, dass sie sich nicht widersprechen.

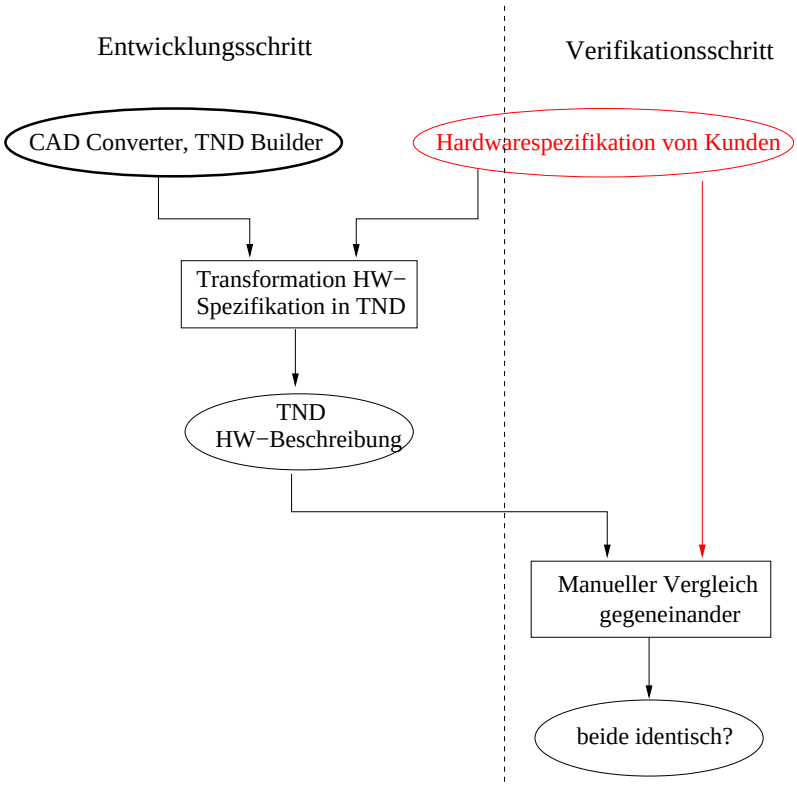


Abbildung 6.4: Validierung der TND-Hardwarebeschreibung

Deng Zhou, Ruben Rothaupt

6.5 Verifikation der Projektierungsdaten

Es muss überprüft werden, ob die Projektierungsdaten eine sichere Steuerung des jeweiligen Gleisnetzes ermöglichen.
Anhand der in der TND-Netzwerkbeschreibung enthaltenen Informationen kann bestimmt werden, an welcher Stelle des Gleisnetzes sich zu welchem Zeitpunkt Bahnen befinden können und welche Sicherheitsbedingungen das System erfüllen muss. Anhand der in den Projektierungsdaten enthaltenen Informationen kann bestimmt werden, wie das Steuerungssystem die Weichen und Signale schalten soll.

Es kann nun durch Model Checking überprüft werden, ob ein Zustand erzeugt werden kann, in dem eine der spezifizierten Sicherheitsbedingungen verletzt wird. Dies wird im Abschnitt 7.6 auf Seite 203 detaillierter beschrieben.

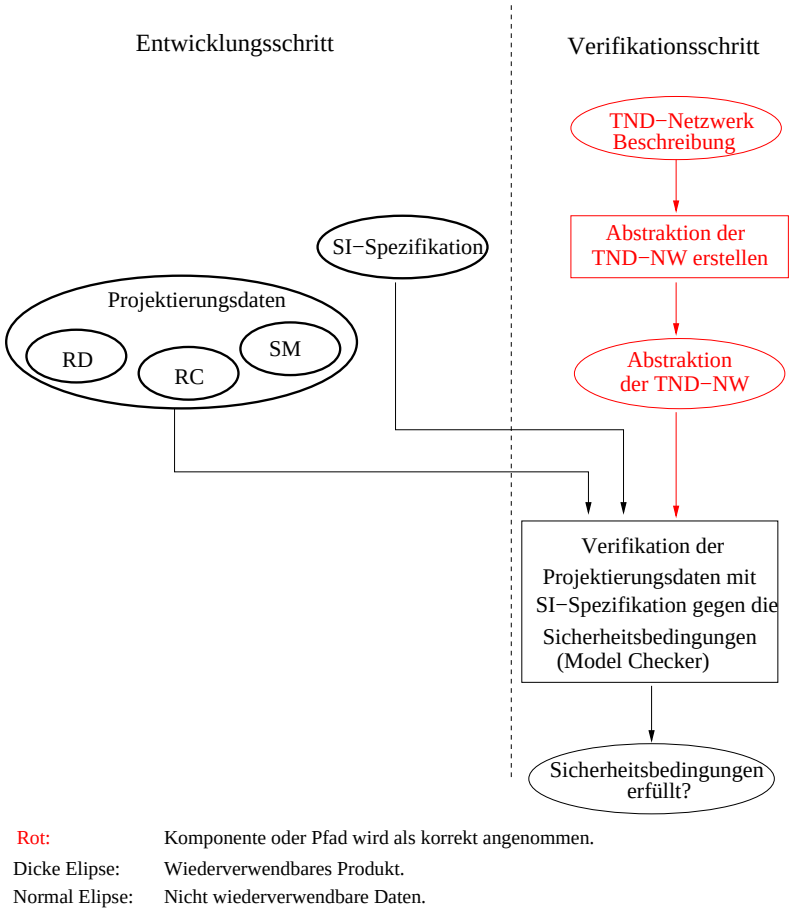


Abbildung 6.5: Verifikation der Projektierungsdaten

Deng Zhou, Ruben Rothaupt

6.6 Test der Steuerungssoftware

Die Steuerungsoftware besteht aus dem wiederverwendbaren Steuerinterpreter und den gleisnetzspezifischen Projektierungsdaten.
Aus der TND und der Softwarespezifikation werden Testprozeduren für die Steuerungssoftware entwickelt. Dafür wird eine formale Spezifikation benötigt, die das zu testende Verhalten der Software beschreibt. Dann wird getestet, ob die Steuerungssoftware die

Spezifikation auch tatsächlich erfüllt. Der Testvorgang wird im Abschnitt 7.7 auf Seite 255 beschrieben.

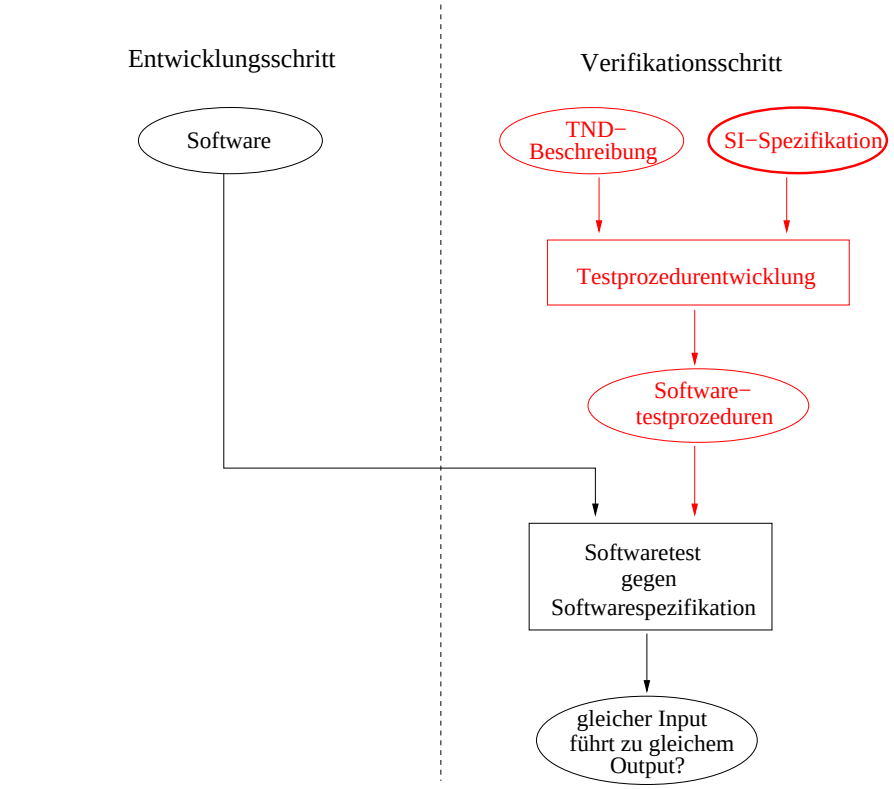


Abbildung 6.6: Softwaretest gegen Softwarespezifikation

Deng Zhou, Ruben Rothaupt

6.7 Test der Treiberbibliothek

Die Treiber übernehmen die Rolle der Schnittstelle zwischen der Software und der Hardware. Es wird eine Bibliothek von allen gegebenen Treibern mit deren jeweiligen Besonderheiten definiert.
Das richtige Verhalten der Treiber wird durch Tests auf allen Integrationsebenen geprüft.

Deng Zhou, Ruben Rothaupt

6.8 Test des fertigen Steuerungssystems

Schließlich muss noch das fertige Steuerungssystem getestet werden. Dabei handelt es sich um das Endprodukt von TRACS.
Aus der TND und der Systemspezifikation werden Tesprozeduren entwickelt und ein Testsystem erstellt. Die korrekte Funktionalität des Steuerungssystems muss auf Grundlage einer formalen Spezifikation nachgewiesen werden. Es muss überprüft werden, ob das ausführbare Steuerungssystem alle seine beabsichtigten Aufgaben erfüllt. Hierauf wird ebenfalls im Abschnitt 7.7 auf Seite 255 näher eingegangen.

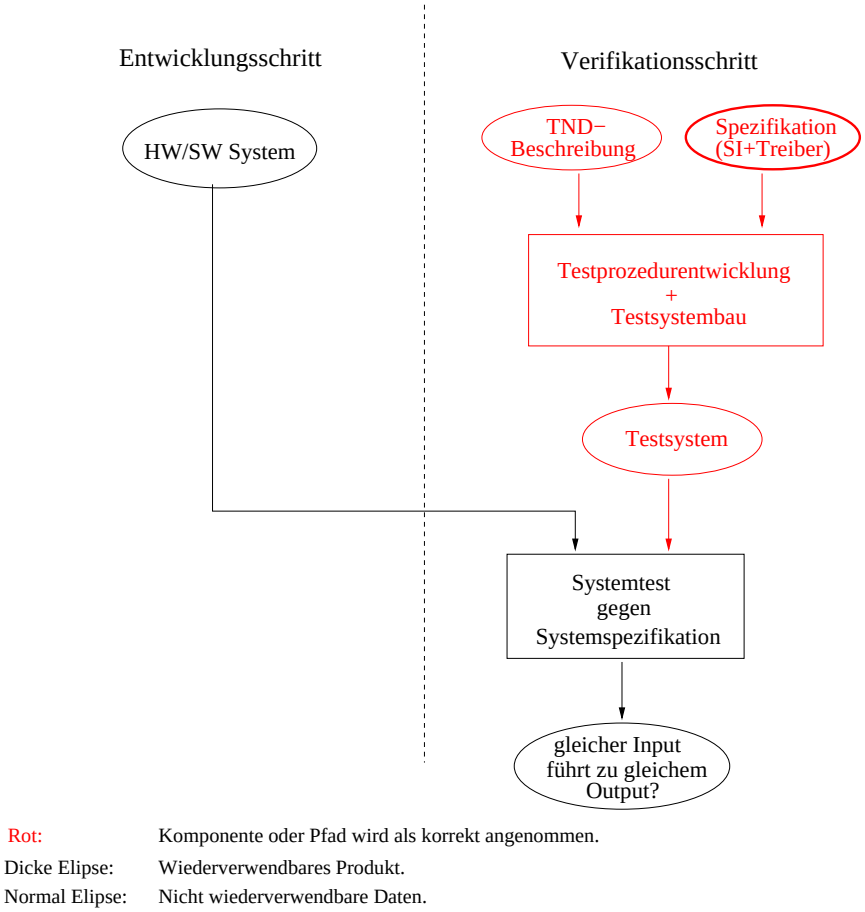


Abbildung 6.7: Systemtest gegen Systemspezifikation

Kapitel 7

Beschreibung der Komponenten

7.1 Tram Network Description - TND

7.1.1 Überblick

Die **Tram Network Description** - kurz **TND** - ist eine sogenannte Domain Specific Language (DSL). Bei DSLs handelt es sich, wie ihr Name schon andeutet, um Sprachen, die speziell für ein bestimmtes Anwendungsgebiet entworfen werden. Diese Tatsache erlaubt es, eine Sprache genau an die Bedürfnisse des jeweiligen Gebietes anzupassen, so dass sie insbesondere für Experten aus dem Gebiet einfach verwendbar ist.
Die Aufgabenstellung des Projektes TRACS verlangt nach einem Beschreibungsformalismus für Straßenbahn-Gleisnetze und der auf einem Gleisnetz aufbauenden Steueraufgabe. Dieser Formalismus wurde in Gestalt der Sprache TND entwickelt. In dieser Sprache können Gleisnetze für Straßenbahnsysteme beschrieben werden, außerdem Informationen über in dem Gleisnetz bestehende Routen, Steuerinformationen für die einzelnen Routen, ggf. Konflikte zwischen Routen, Hardwaredaten. Damit können in einem spezifischen TND-Dokument alle zur Erfüllung der Steueraufgabe nötigen Daten angelegt werden.
Aus einer in TND vorliegenden Beschreibung einer Steueraufgabe wird dann das ausführbare Steuersystem generiert. Dazu wird die Beschreibung in ein Binärformat umgewandelt, welches dann von einem Steuerinterpreter verarbeitet wird.
Das TND-Format ist unabhängig von etwaigen Eingabetools gehalten. TND-Dateien können prinzipiell „per Hand“ in einem Texteditor erstellt werden. Im Rahmen von TRACS wurden aber auch mehrere Eingabehilfen hergestellt. Zum einen handelt es sich hier um einen CAD-TND-Konverter, der aus CAD-Zeichnungen von Gleisnetzen TND-Dateien erzeugen kann. Die so erzeugten Dateien erhalten dann zwar alle Daten über das physisch vorhandene Gleisnetz an sich, jedoch noch nicht die zur Steuerung erforderlichen Informationen über Routen, Konflikte etc. An dieser Stelle setzt der ebenfalls von TRACS entwickelte TND-Builder an, der eine Eingabeoberfläche für eben diese

noch fehlenden Daten bereitstellt, so dass die in der TND abzulegenden Informationen vervollständigt werden können.

Arne Stahlbock

7.1.2 Beschreibungssprache TND

Die TND-Beschreibungen enthalten folgende Informationen:

- Netzwerk
- Verschlusstabellen
- Hardware-Beschreibung

7.1.2.1 Netzwerk

Das Netzwerk besteht aus:

- Weichen - **points**
- Signalen - **signals**
- Sensoren - **sensors**
- Streckensegmente - **relations**
- Kreuzungen - **crossings**

Darüber hinaus gibt es Kreuzungsweichen - **slipswitches**. Diese bestehen jedoch direkt aus Kreuzungen und Weichen, so dass sie zunächst nicht als eigenes Hardware-Element geführt werden, sondern diejenigen Weichen und Kreuzungen, die jeweils zu einer Kreuzungsweiche zusammengeschlossen sind, an einer bestimmten Stelle in der Beschreibung als solche deklariert werden.
Die Netzwerkbeschreibung stellt die Zusammenhänge zwischen den vorgenannten Elementen her.
Eine Erweiterung des Netzwerkes stellt der Netzgraph dar. Dieser liegt häufig in visueller Form (CAD-Darstellung) vor und enthält zusätzlich topologische Informationen, wie zum Beispiel Koordinaten (**coordinates** g200:(0,0,0)) und Markierungspunkte (**marks** x100).

7.1.2.2 Verschlusstabellen

Verschlusstabellen beschreiben die Konditionen, die notwendig sind, um die sichere Durchfahrt von Bahnen durch das Gleisnetz zu gewährleisten. Sie bestehen aus:

- **Route Definition Table**: Definiert die Routen, die die Bahnen durch das Netz zurücklegen sollen, über die Reihenfolge der zu überfahrenden Sensoren.
- **Conflict Table**: Identifiziert die Konflikte zwischen Routen. Dabei unterscheidet man zwischen *point-conflicts* - schließen sich durch unterschiedliche Weichenstellungen aus und *conflicts* - anderweitige Ausschlüsse.
- **Point Position Table**: Definiert die Positionen der Weichen, die für die jeweilige Route nötig sind.
- **Signal Setting Table** : Enthält die entsprechende Richtungsangabe des Eingangssignals der Route, um die Einfahrt freizugeben.

Die Verschlusstabellen beinhalten also diejenigen Informationen, die ein Steuerungssystem benötigt, um zu wissen, welche Routen es gibt, welche Weichen- und Signalstellungen für welche Route geschaltet werden müssen und um zu entscheiden, welche Route wann freigegeben werden kann.

7.1.2.3 Hardware-Beschreibung

Die im Gleisnetz vorkommenden Hardware-Elemente müssen bezüglich ihrer Funktionalität näher beschrieben werden, um die Steueraufgabe ausführen zu können. Benötigt werden die folgenden Informationen:

- für Weichen: Richtungen, die die Weiche schalten kann
- für Weichen: Zeit, die nach einer Schaltanforderung maximal vergehen darf, bis die Schaltung erfolgt ist
- für Weichen: zusätzliche Eigenschaften wie aktiv/passiv, Rückfallweiche ja/nein (und in welche Richtung?), auffahrbar ja/nein
- für Weichen und Kreuzungen: Information, ob und mit welchen anderen Elementen zusammen ein Element eine Kreuzungsweiche bildet
- für Signale: Richtungen, die das Signal freigeben kann
- für Signale: Zeit, die nach einer Schaltanforderung maximal vergehen darf, bis die Schaltung erfolgt ist

- für Signale: zusätzliche Schaltzustände, die ein Signal einnehmen kann (Warteanzeigen, „A“-Anzeigen)
- für Sensoren: Typ des Sensors (Toggle, Toggle mit Richtungserkennung, State, State mit Richtungserkennung, Route Request)
- für Weichen, Signale, Sensoren und Kreuzungen (letztere nur sofern in Kreuzungsweiche enthalten): Information über die Zugehörigkeit zu Hardwareklassen, um eine Zuordnung herstellen zu können, mit welchem Treiber ein jeweiliges Element anzusteuern ist

Susanne Spiller, Arne Stahlbock

7.1.3 Sprachdefinition der TND

Eine typische Sprachbeschreibung besteht aus der lexikalischen Grammatik, der syntaktischen Grammatik, Typregeln und der Semantik. Im Folgenden werden diese Elemente für die Sprache TND dargestellt.

7.1.3.1 Lexikalische Grammatik

In der lexikalischen Grammatik wird definiert, welche bestimmten Folgen von Einzelzeichen in einer Sprache Symbole bilden.

Für die Sprache TND ist das relativ unspektakulär, es gibt eine gewisse Menge an genau festgelegten Schlüsselworten, darüber hinaus können aber auch einige Worte beliebig, nur mit der Vorgabe eines Anfangsbuchstabens, gebildet werden, diese dienen später als Bezeichner. Außerdem zählen Ganzzahlen sowie einige Sonderzeichen zum Sprachumfang. Als Whitespace gelten Leerzeichen, Tabulator und Zeilenumbruch.

Die komplette lexikalische Grammatik ist im Anhang A.1 auf Seite 453 aufgeschrieben.

7.1.3.2 Syntaktische Grammatik und Semantik

Die syntaktische Grammatik definiert, welche Folgen von Symbolen (die ihrerseits also in der lexikalischen Grammatik definiert werden) in einer Sprache gültig sind.

In der Folge wird die syntaktische Grammatik für die Sprache TND in EBNF-Form dargestellt und erläutert. Eine zusammenhängende und vollständige TND ist in den Anhängen A.1 auf Seite 453 und A.2 auf Seite 456 enthalten.

Anmerkung: Die folgende Beschreibung der EBNF der TND wird anhand von Beispielen verdeutlicht. Eine Zeichung des Beispielgleisnetzes ist unter 7.1 zu sehen.

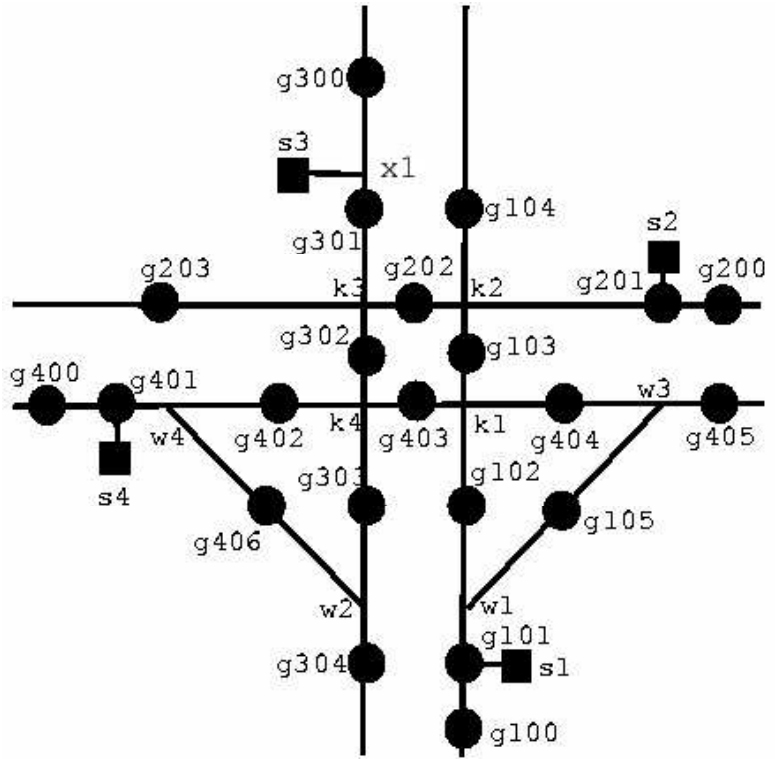


Abbildung 7.1: Beispielgleisnetz

7.1.3.2.1 Gesamtaufbau

```
<tnd>           ::= <defblock> <coordblock> <sigpropblock>
                    [<ptpropblock>] [<slipswitchblock>]
                    <relblock> <sigposblock>
                    <routeblock> <condblock> <clearblock>
                    [<ptconfblock>] [<confblock>] <hwblock>
```

Die TND enthält verschiedene Blöcke, in denen die Beschreibung des Gleisnetzes vorgenommen wird. Die Reihenfolge dieser Blöcke ist fest vorgegeben, dabei sind einige Blöcke optional.

7.1.3.2.2 DefBlock

```
<defblock>      ::= <DEFINITIONS> "{" {<typdef>} "}"
<typdef>        ::= <sensdef> | <pointdef> | <sigdef> |
                    <routedef> | <markdef> | <hwdef> |
                    <crossdef>
```

```
<sensdef>      ::= <senstype> ":" <SENSID> {"," <SENSID>} ";"
<senstype>     ::= <RRSENSORS> | <TDSENSORS> | <TGSENSORS> |
                   <SDSENSORS> | <SGSENSORS>

<pointdef>     ::= <pointtype> ":" <POINTID> {"," <POINTID>} ";"
<pointtype>    ::= <SLPOINTS> | <SRPOINTS> | <LRPOINTS> |
                   <SLRPOINTS>

<sigdef>       ::= <sigtype> ":" <SIGID> {"," <SIGID>} ";"
<sigtype>      ::= <LSIGNALS> | <SSIGNALS> | <RSIGNALS> |
                   <SLSIGNALS> | <SRSIGNALS> | <LRSIGNALS> |
                   <SLRSIGNALS>

<routedef>     ::= <ROUTES> ":" <ROUTEID> {"," <ROUTEID>} ";"

<markdef>      ::= <MARKS> ":" <MARKID> {"," <MARKID>} ";"

<hwdef>        ::= <HWCLASSES> ":" <HWID> {"," <HWID>} ";"

<crossdef>     ::= <CROSSINGS> ":" <CROSSID> {"," <CROSSID>} ";"
```

Im Definitionsblock werden alle in der Gleisnetzbeschreibung vorkommenden Elemente deklariert. Das gilt für Hardware-Elemente wie Weichen, Sensoren, Signale und Kreuzungen genau wie für logische Elemente (Routen, Markierungen und Hardwareklassen).

Dazu wird eine beliebige Anzahl an Deklarationen vorgenommen, in denen jeweils ein Elementtyp genannt wird, danach die Bezeichner für alle vorkommenden Elemente dieses Typs. Es wird dabei nach verschiedenen Arten von Elementen unterschieden (Sensoren, Weichen, Signale, Routen, Markierungen, Kreuzungen, Hardwareklassen), dabei ist für deren jeweilige IDs ein bestimmter Anfangsbuchstabe vorgegeben.

Bei den Sensoren wurden die Typen an die im Weltmodell beschriebenen Typen angepasst, also `rr` (route-request), `td` (Toggle mit Richtungserkennung), `tg` (Toggle ohne Richtungserkennung), `sd` (Zustand mit Richtungserkennung) und `sg` (Zustand ohne Richtungserkennung). Bezeichner für Sensoren beginnen immer mit `g`. Näheres zu den Typen im entsprechenden Abschnitt des Weltmodells (2.3.2.2 auf Seite 21).
Bei den Weichen gibt es zwei grundlegende Typen, nämlich Zwei- und Dreiwegeweichen. Wir unterscheiden die Weichen nach den Richtungen, die sie annehmen können: z. B. `sl-points` oder `slr-points`. Weichenbezeichner beginnen mit `w`. Näheres zu Weichen in 2.3.2.1.2 auf Seite 20.

Für Signale sind zahlreiche Typen vorgesehen, die Buchstaben l, s und r in den Typbezeichnungen stehen dabei für die Richtungen, in die das Signal die Fahrt freigeben kann. Ein `lr-signal` beispielsweise kann demnach STOP, LEFT und RIGHT anzeigen. Signalbezeichner beginnen mit `s`. Näheres zu Signalen in 2.3.2.3 auf Seite 22.

Die Deklaration der Routen ist so zu verstehen, dass für die später genau spezifizierten Routen Bezeichner reserviert werden. Diese Bezeichner beginnen mit `r`.

Ähnlich verhält es sich mit den Markierungen (marks), hierbei handelt es sich um Hilfspunkte auf dem Gleis, die bei der Transformation in eine graphische Darstellung benötigt werden. In diesem Block werden zunächst nur Bezeichner eingeführt.

Ebenso werden Kreuzungen an dieser Stelle deklariert.

Weiterhin werden Bezeichner für Hardwareklassen eingeführt, die für den Steuerinterpreter von Interesse bezüglich der Treiberansteuerung sind.

Beispiel:

```
definitions {
    sd-sensors: g101, g102, g103, g104, g105, g201, g202, g203, g301,
                g302, g303, g304, g401, g402, g403, g404, g405, g406;
    rr-sensors: g100, g200, g300, g400;
    s-signals:  s2, s3;
    sr-signals: s1, s4;
    sl-points: w2, w3;
    sr-points: w1, w4;
    routes: route1, route2, route3, route4, route5, route6;
    marks: x1;
    hwclasses: c1, c2;
}
```

Das Beispielnetz enthält eine gewisse Anzahl an Sensoren (als Typ wurde ohne Beschränkung der Allgemeinheit `sd` gewählt - es hätte auch durchaus ein anderer sein können), vier Signale unterschiedlichen Typs, vier Zweiwegeweichen und es sind sechs Routen definiert, wie das Netz zu durchfahren ist. Außerdem gibt es genau eine Markierung.

7.1.3.2.3 CoordBlock

```
<coordblock>   ::= <COORDINATES> "{" {<coorddef>} "}"
<coorddef>     ::= <elementid> ":" "(" <COORD> ","
                   <COORD> "," <COORD> ")" ";"
<elementid>    ::= <SENSID> | <POINTID> | <SIGID> | <MARKID> | <CROSSID>
```

Im CoordBlock findet die Zuordnung der Hardwareelemente (Weichen, Signale, Sensoren, Kreuzungen) und Markierungen zu ihren Ortskoordinaten statt. Er beginnt mit dem Schlüsselwort `coordinates` und umfasst beliebig viele Koordinaten-Definitionen in einem von geschweiften Klammern umfassten Block. Eine einzelne Definition enthält: den Bezeichner des Elements, gefolgt von einem Doppelpunkt, einer geöffneten Klammer, dann drei jeweils durch Komma getrennte Ganzzahlen, geschlossene Klammer und Semikolon. Die drei Zahlen sind als x-, y- und z-Koordinaten zu interpretieren. Der Nullpunkt des Koordinatensystems muss nicht zwingend gegeben sein, da die Koordinaten lediglich dazu dienen, die Relation zwischen den einzelnen Elementen darzustellen. Als Maßeinheit wird Zentimeter angenommen, daher erscheinen Integerzahlen als ausreichend präzise. (Das folgende Beispiel wäre dann ein mikroskopisches Gleisnetz, es dient ja aber auch nur der Verdeutlichung der Notation.) Koordinaten sind für alle tatsächlich auf/am Gleis befindlichen Elemente (Sensoren, Weichen, Kreuzungen) anzugeben, bei Signalen und Markierungen ist der punktgenaue Standort nur wichtig, falls dies für eine Visualisierung als notwendig erachtet wird.

Die Koordinatenangaben sind wie folgt zu verstehen:

- für Sensoren: ein Punkt, der sich genau in der Mitte zwischen den beiden zu einem Gleis gehörenden Schienenstränge befindet, in Längsrichtung an der Stelle, die eine Bahn erreichen muss, um eine Auslösung des Sensors zu bewirken
- für Weichen: ebenfalls ein Punkt in der Mitte zwischen den zwei Schienensträngen, in Längsrichtung an der Stelle, an der die Verzweigung beginnt bzw. endet (auf Höhe dieses Punktes wechseln die Räder der Bahn das Gleis)
- für Signale: der Punkt, an dem das Signal steht (an dem der Signalmast aus dem Boden kommt) - handelt es sich um ein hängendes Signal, dann die Projektion des Signals auf den Boden
- für Kreuzungen: der Mittelpunkt der Kreuzung
- für Markierungen: ein Punkt in der Mitte zwischen den zwei Schienensträngen

Beispiel:

```
coordinates {
    g100:(5000, 8500, 0);
    g101:(5000, 8000, 0);
    g102:(5000, 6000, 0);
    g103:(5000, 4000, 0);
    g104:(5000, 2000, 0);
    g105:(6000, 6000, 0);
    g200:(8000, 3200, 0);
    g201:(7000, 3200, 0);
    g202:(4000, 3200, 0);
    g203:(1000, 3200, 0);
    g300:(3000,  500, 0);
    g301:(3000, 2000, 0);
    g302:(3000, 4000, 0);
    g303:(3000, 6000, 0);
    g304:(3000, 8000, 0);
    g400:(   0, 4800, 0);
    g401:( 500, 4800, 0);
    g402:(2000, 4800, 0);
    g403:(4000, 4800, 0);
    g404:(6000, 4800, 0);
    g405:(8000, 4800, 0);
    g406:(2000, 6000, 0);
    w1:(5000, 7500, 0);
    w2:(3000, 7500, 0);
    w3:(7000, 4800, 0);
    w4:(1000, 4800, 0);
    s1:(5200, 8000, 0);
    s2:(7000, 3000, 0);
    s3:(2800, 2000, 0);
    s4:( 500, 5000, 0);
    k1:(5000, 4800, 0);
    k2:(5000, 3200, 0);
    k3:(3000, 3200, 0);
    k4:(3000, 4800, 0);
    x1:(3000, 1500, 0);
```

(Die Koordinaten sind sicher nicht maßstabsgetreu.)

7.1.3.2.4 SigPropBlock

```
<sigpropblock> ::= <SIGNAL_PROPERTIES> "{" {<sigpropdef>} "}"
<sigpropdef>   ::= <SIGID> ":" [<WAIT_STRAIGHT>] [<WAIT_LEFT>]
                   [<WAIT_RIGHT>] [<RR_STRAIGHT>] [<RR_LEFT>]
                   [<RR_RIGHT>] <SWITCHTIME> <POSINT> ";"
```

Dieser Block enthält zu jedem im DefBlock aufgeführten Signal eine nähere Eigenschaftsbeschreibung. Er beginnt mit dem Schlüsselwort `signal-properties`, gefolgt von einer in geschweifte Klammern eingeschlossenen, beliebigen Anzahl von Eigenschaftsbeschreibungen. Diese wiederum bestehen aus einem Signalbezeichner, gefolgt von einem Doppelpunkt, auf den eine Reihe Schlüsselworte folgen kann und am Ende das Schlüsselwort `switchtime` mit einer positiven Ganzzahl folgen muss. Die einzelne Eigenschaftsbeschreibung endet mit einem Semikolon.
Bedeutung:
Jedes der optional vorkommenden Token wie WAIT_STRAIGHT etc. gibt an, ob ein Signal über die entsprechende Anzeigemöglichkeit verfügt. Kommt ein Token vor, ist die entsprechende Anzeigemöglichkeit am Signal vorhanden, ansonsten nicht. Verpflichtend ist die Switchtime in Millisekunden anzugeben.

Beispiel:

```
signal-properties {
    s1: rr-straight rr-right switchtime 100;
    s2: wait-straight switchtime 200;
    s3: switchtime 200;
    s4: switchtime 100;
}
```

7.1.3.2.5 PointPropBlock

```
<ptpropblock>  ::= <POINT_PROPERTIES> "{" {<ptpropdef>} "}"
<ptpropdef>    ::= <POINTID> ":" [<PASSIVE>] [<FALLBACK> <direction>]
                   [<BREAKABLE>] <SWITCHTIME> <POSINT> ";"
<direction>    ::= <LEFT> | <STRAIGHT> | <RIGHT>
```

Der Aufbau dieses Blockes ist analog zum vorigen, er unterscheidet sich durch das einleitende Schlüsselwort `point-properties`, außerdem handelt es sich bei den hier aufgeführten Eigenschaftsbeschreibungen um solche für Weichen, die daher auch mit Weichenbezeichnern beginnen und andere optionale Schlüsselworte haben. Die Switchtime jedoch ist wieder genau wie oben.
Bedeutung:

PASSIVE: Die Weiche hat keine automatische Stellvorrichtung. Fehlt das Token bei einer Weiche, ist eine solche vorhanden.
FALLBACK: Die Weiche ist eine Rückfallweiche (d.h. fällt nach dem Auffahren aus der nicht geschalteten Richtung in ihre vorherige Lage zurück). Diese Standardlage muss angegeben werden (bei Aktivweichen kann sie durch den Motor umgestellt werden).
BREAKABLE: Die Weiche wird durch Auffahren aus der falschen Richtung potentiell beschädigt (= darf nicht aufgefahren werden).
Verpflichtend ist die Switchtime in Millisekunden anzugeben.

Beispiel:

```
point-properties {
    w1: switchtime 500;
    w2: passive fallback straight switchtime 500;
    w3: passive switchtime 800;
    w4: passive switchtime 800;
}
```

7.1.3.2.6 SlipswitchBlock

```
<slipswitchblock> ::= <SLIP_SWITCHES> "{" {<slipswitchdef>} "}"
<slipswitchdef>   ::= <CROSSID> ":" <POINTID> "," <POINTID>
                      ["," <POINTID> "," <POINTID>]
                      ":" <SWITCHTIME> <POSINT> ";"
```

Dieser Block wird durch das Schlüsselwort `slip-switches` eingeleitet, gefolgt von einer in geschweifte Klammern eingeschlossenen beliebigen Anzahl von Kreuzungsweichen-Definitionen. Eine solche beginnt mit einem Kreuzungsbezeichner, der nach einem Doppelpunkt von zwei oder vier durch Komma getrennten Weichenbezeichnern gefolgt wird, bevor wiederum ein Doppelpunkt, das Schlüsselwort `switchtime`, eine positive Ganzzahl und ein abschließendes Semikolon folgt.
Zu verstehen ist eine einzelne Kreuzungsweichen-Definition so, dass hier eine Kreuzung mit zwei oder vier Teilweichen so gekoppelt ist, dass sie ein einzelnes Element, eine Kreuzungsweiche bildet, das nur von einer Bahn gleichzeitig befahren werden kann. Eine solche Kreuzungsweiche hat dann auch eine gemeinsame Schaltzeit. Sollten für die Teilweichen unterschiedliche Schaltzeiten vorliegen, so ist hier die höchste von ihnen zu nennen (nach dieser Zeit muss das gesamte Element auf jeden Fall geschaltet haben).
Ein Beispiel muss an dieser Stelle entfallen, da im Beispielnetz kein solches Element vorhanden ist.

7.1.3.2.7 RelBlock

```
<relblock>       ::= <RELATIONS> "{" {<reldef>} "}"
<reldef>         ::= <relation> | <relnone>
<relation>       ::= <elementid-side> ":" {<MARKID> ","}
                     <elementid-side> ";"
<relnone>        ::= <sensid-side> ":" {<MARKID> ","} <een> ";"
<een>            ::= <ENTRANCE> | <EXIT> | <NONE>
<elementid-side> ::= <sensid-side> | <crossid-side> | <pointid-side>
<sensid-side>    ::= <SENSID> <SIDE>
<crossid-side>   ::= <CROSSID> <cross-side>
<cross-side>     ::= <SIDE> | <CROSS_SIDE>
<pointid-side>   ::= <POINTID> <POINT_SIDE>
```

Der RelBlock gibt Nachbarschaftsbeziehungen zwischen Elementen an (und beschreibt diesbezüglich Gleisstücke): Schlüsselwort dieses Blocks ist `relations`. Nach bekanntem Muster gibt es dann eine beliebige Anzahl von Relationsbeschreibungen, von denen eine einzelne in verschiedenen Formen auftreten kann. Ein Element und eine Seitenangabe, gefolgt von einem Doppelpunkt, bilden den Anfang, danach kommt eine (evtl. leere) Liste von Marks, dann ein weiteres Element mit Seitenangabe. Eine weitere Form enthält einen Sensor mit einer Seitenangabe, wieder gefolgt von einer ggf. leeren Mark-Liste, dann eines der Schlüsselworte `entrance`, `exit` oder `none`.
Bedeutung:
Für den Fall, dass zwei Elemente mit ihren jeweiligen Seiten genannt werden, heißt dies, dass die beiden Elemente mit ihren genannten Seiten einander zugewandt sind, mithin durch ein Gleisstück verbunden sind. Eine Bahn, die das eine Element zu der dort genannten Seite hin verlässt, wird also als nächstes das zweite Element an dessen genannter Seite erreichen. Zwischen den Elementen kann eine Folge von Markierungen vorliegen, die auch in der Reihenfolge vom erstgenannten Element zum zweiten angeordnet sind.
Bei Sensoren gibt es nur zwei Seiten, daher ist hier die Bedeutung relativ simpel: Eine Bahn, die von einer Seite auf einen Sensor fährt, wird ihn an der anderen wieder verlassen. Bei Kreuzungen ist es etwas komplexer, diese haben vier Seiten, die Bedeutung dieser Seiten ist wie folgt: Eine Bahn, die eine Kreuzung von Seite A her befährt, wird an Seite B ausfahren - analog sind C und D verbunden. Für die Weiche wurden der Anschaulichkeit halber andere Buchstaben gewählt, und zwar sind die bis zu drei Seiten auf der stumpfen Seite mit L, S und R bezeichnet (für left, straight, right), die spitze Seite trägt die Bezeichnung Y.
Im zweiten Fall, dass also ein Sensor mit Seite, gefolgt von einer Mark-Liste und einem der erwähnten Schlüsselworte, auftritt, liegt dort ein Gleisnetzende vor (bei leerer Mark-Liste direkt an der genannten Seite des ersten Sensors, bei nicht leerer Mark-Liste an der letzen Mark). Die Art des Gleisendes wird durch das Schlüsselwort bestimmt:
ENTRANCE: hier kann eine Bahn in das Gleisnetz gelangen oder es verlassen

EXIT: hier kann eine Bahn das Gleisnetz verlassen, aber nicht hineingelangen
NONE: hier keine Ein- oder Ausfahrt (also z.B. Prellbock, totes Ende etc.)

Beispiel:

```
relations {
    g100 A: entrance;
    g200 A: entrance;
    g300 A: entrance;
    g400 A: entrance;
    g100 B: g101 A;
    g200 B: g201 A;
    g300 B: x1, g301 A;
    g400 B: g401 A;
    g104 B: exit;
    g203 B: exit;
    g304 B: exit;
    g405 B: exit;
    k1 A: g102 B;
    k1 B: g103 A;
    k1 C: g403 B;
    k1 D: g404 A;
    k2 A: g103 B;
    k2 B: g104 A;
    k2 C: g201 B;
    k2 D: g202 A;
    k3 A: g301 B;
    k3 B: g302 A;
    k3 C: g202 B;
    k3 D: g203 A;
    k4 A: g402 B;
    k4 B: g403 A;
    k4 C: g302 B;
    k4 D: g303 A;
    w1 Y: g101 B;
    w1 S: g102 A;
    w1 R: g105 A;
    w2 Y: g304 A;
    w2 S: g303 B;
    w2 L: g406 B;
    w3 Y: g405 A;
```

```
    w3 S: g404 B;
    w3 L: g105 B;
    w4 Y: g401 B;
    w4 S: g402 A;
    w4 R: g406 A;
}
```

7.1.3.2.8 SigPosBlock

```
<sigposblock>  ::= <SIGPOSITIONS> "{" {<sigposdef>} "}"
<sigposdef>    ::= <SIGID> ":" [<sensid-side> "-" <MARKID> "-"]
                   <sensid-side> ";"
```

Hier werden die Positionen der Signale im Gleisnetz beschrieben, Blockschlüsselwort ist `signal-positions`. Genannt ist die Signal-ID, danach gibt es zwei Möglichkeiten:
a) Das Signal steht direkt an einem Sensor. In diesem Fall ist nur die Sensor-ID mit der Seite genannt, in die auch das Signal weist. D.h. eine von dieser Seite auf den Sensor zufahrende Bahn muss das Signal beachten.
b) Das Signal steht zwischen zwei Sensoren. In diesem Fall muss es an einer Markierung stehen, damit der genaue Punkt auf dem Gleis, an dem eine Bahn ggf. halten muss, bekannt ist. Aufgeführt werden hier in diesem Fall der Sensor (+Seite), von dem die Bahn kommt, die Markierung, an der das Signal steht und der Sensor mit der Seite, auf die die Bahn zufährt.
Beiden Darstellungen ist also gemeinsam, dass immer der letzte Sensor derjenige ist, auf den die Bahn aus der für ihn angegebenen Seite zufahren muss, um das Signal zu sehen.

Beispiel:

```
signal-positions {
    s1: g101 A;
    s2: g201 A;
    s3: g300 B - x1 - g301 A;
    s4: g401 A;
}
```

7.1.3.2.9 RouteBlock

```
<routeblock>   ::= <ROUTEDEFINITIONS> "{" {<rtdef>} "}"
<rtdef>        ::= <ROUTEID> ":" <SENSID> "," <SENSID> {"," <SENSID>}
                   "-" <REQUESTAT> ":" <SENSID> {"," <SENSID>} ";"
```

Vor dem RouteBlock steht das Wort `routes`, welches den Block identifiziert. Der gesamte Block von beliebig vielen Routendefinitionen wird mit geschweiften Klammern umrahmt. Jede Routendefinition fängt mit dem Namen der Route an, auf den ein Doppelpunkt folgt. Anschließend werden alle Sensoren dieser Route aufgezählt und durch Kommata getrennt. Es müssen mindestens zwei Sensoren pro Route angegeben werden. Es folgt ein Bindestrich, das Schlüsselwort `request-at` und eine weitere Liste von Sensoren Jede Beschreibung einer einzelnen Route endet mit einem Semikolon.
Bedeutung:
Der RouteBlock enthält die Informationen über die zu den einzelnen Routen gehörenden Sequenzen von Sensoren, wie sie auch in einer *Route Definition Table* vorhanden sind. Zu jeder Route wird hier die Reihenfolge der Sensoren angegeben, in der die Sensoren bei der Befahrung der Route überfahren werden. Außerdem werden die RR-Sensoren genannt, an denen die Route angefordert werden kann (die zweite Sensorenliste).

Beispiel:

```
routes {
    route1 : g101 , g102 , g103 , g104 - request-at g100;
    route2 : g201 , g202 , g203 - request-at g200;
    route3 : g301 , g302 , g303 , g304 - request-at g300;
    route4 : g401 , g402 , g403 , g404 , g405 - request-at g400;
    route5 : g101 , g105 , g405 - request-at g100;
    route6 : g401 , g406 , g304 - request-at g400;
}
```

Dieses sind die Routenbeschreibungen des oben genannten Beispieles in Abbildung 7.1 auf Seite 63. Man kann u.a. erkennen, dass `route1` und `route5` am gleichen Sensor beginnen und sich dann nach dem Sensor `g101` trennen. Folglich muss eine Weiche zwischen `g101` und den Sensoren `g102` bzw. `g105` sein, wie auch in der Abbildung zu sehen ist.

7.1.3.2.10 CondBlock

```
<condblock>     ::= <CONDITIONS> "{" {<conddef>} "}"
<conddef>       ::= <ROUTEID> ":" <condition>
                    {"," <condition>} ";"
<condition>     ::= <POINTID> <direction>
```

Vor dem CondBlock steht das Wort `conditions`, welches den Block identifiziert. Der gesamte Block von beliebig vielen Einfahrtsbedingungen wird mit geschweiften Klammern umrahmt. Jede Einfahrtsbedingung fängt mit dem Namen der Route an, auf den ein Doppelpunkt folgt. Anschließend werden alle Einfahrtsbedingungen, getrennt durch

Kommata, angegeben. Jede Einfahrtsbedingungsbeschreibung einer einzelnen Route endet mit einem Semikolon. Eine Einfahrtsbedingung besteht aus dem Weichenbezeichner und der geforderten Weichenstellung. Allgemein mögliche Weichenstellungen sind `left`, `right` und `straight`.

Der CondBlock beinhaltet im Prinzip die Information der *Request Condition* – also der Einfahrtsbedingungen für die einzelnen Routen. Es werden die Weichen angegeben, über die die Route führt und die Stellungen der Weichen, die diese haben müssen, damit die Bahn die geforderte Route befahren kann.

Beispiel:

```
conditions {
    route1 : w1 straight;
    route4 : w4 straight;
    route5 : w1 right;
    route6 : w4 right;
}
```

Dieses sind die Einfahrtsbedingungen des oben genannten Beispieles in Abbildung 7.1 auf Seite 63. Die Route `route1` und die Route `route5` trennen sich folglich bei Weiche `w1`.

7.1.3.2.11 ClearBlock

```
<clearblock>   ::= <CLEARANCES> "{" {<cleardef>} "}"
<cleardef>     ::= <ROUTEID> ":" <clearance>
                   {"," <clearance>} ";"
<clearance>    ::= <SIGID> <direction>
```

Vor dem ClearBlock steht das Wort `clearances`, welches den Block identifiziert. Der gesamte Block von beliebig vielen Freigabesignalen wird mit geschweiften Klammern umrahmt. Jede Freigabedefinition beginnt mit einem Routenbezeichner, gefolgt von einem Doppelpunkt. Anschließend werden alle Freigabebedingungen, getrennt durch Kommata, angegeben. Jede Beschreibung der Freigabesignale einer Route endet mit einem Semikolon. Eine Freigabebedingung besteht aus dem Signalbezeichner und der geforderten Signalstellung. Allgemein mögliche Signalstellungen sind `left`, `right` und `straight`.

Der ClearBlock enthält die Informationen der *Signal Setting Table*. Zu jeder Route werden hier die Signale und ihre Signalstellungen angegeben, die nötig sind, um die Strecke freizuschalten und befahren zu können. Bei den Signalen handelt es sich um das Eingangssignal der Route und die weiteren Signale, die z. B. vor Weichen stehen. Eine Route besteht aus mindestens einem Freigabesignal.

Beispiel:

```
clearances {
    route1 : s1 straight ;
    route2 : s2 straight ;
    route3 : s3 straight ;
    route4 : s4 straight ;
    route5 : s1 right ;
    route6 : s4 right ;
}
```

Dieses sind die Signalstellungen für die einzelnen Routen des oben genannten Beispieles in Abbildung 7.1 auf Seite 63. Sie entsprechen genau den Richtungen der Weichenstellungen im CondBlock (s. oben).

7.1.3.2.12 PtConfBlock

```
<ptconfblock>  ::= <POINT_CONFLICTS> "{" {<confdef>} "}"
<confdef>      ::= <ROUTEID> ":" <ROUTEID> {"," <ROUTEID>} ";"
```

Vor dem PtConfBlock steht das Wort `point-conflicts`, welches den Block identifiziert. Der gesamte Block von beliebig vielen Konfliktdefinitionen wird mit geschweiften Klammern umrahmt. Jede Konfliktdefinition beginnt mit einem Routenbezeichner, hinter dem ein Doppelpunkt steht. Anschließend werden alle Routen angegeben, die in einem harten Konflikt stehen. Es muss mindestens eine Konfliktroute angegeben und mehrere Routen durch Kommata getrennt werden.
Der PtConfBlock enthält einen Teil der Informationen der *Route Conflict Table*. Hier werden die harten Konflikte beschrieben – in den Tabellen sind diese mit schwarz ausgefüllten Punkten markiert. Diese harten Konflikte treten auf, wenn sich mindestens zwei verschiedene Routen an einer Weiche trennen. Diese Routen werden also aus der gleichen Richtung angefahren und trennen sich dann. Da die Weiche nur eine Stellung gleichzeitig einnehmen kann und ein Umstellen während der Durchfahrt eines Zuges höchst dramatische Folgen hätte, dürfen zwei in einem harten Konflikt stehende Routen nicht gleichzeitig freigeschaltet werden.

Die Angabe der Konflikte erfolgt doppelt, bei jeder der an einem Konflikt beteiligten Routen ist die zweite Konfliktroute angegeben. Dies soll dem Zweck dienen, alle zu einer Route auftretenden Konflikte schnell auffinden zu können – bei einer lediglich einseitigen Angabe müsste der komplette Block nach Konflikten durchsucht werden, so ist es nur eine Zeile. Dies führt zu erhöhter Fehleranfälligkeit bei der Eingabe, eine Fehlererkennung müsste allerdings durch den Compiler möglich sein.

Beispiel:

```
point-conflicts {
    route1 : route5 ;
    route4 : route6 ;
    route5 : route1 ;
    route6 : route4 ;
}
```

Dieses sind die harten Konflikte der einzelnen Routen des oben genannten Beispieles in Abbildung 7.1 auf Seite 63. Da die Route `route1` und die Route `route2` sich an der Weiche `w102` trennen, gibt es einen harten Konflikt zwischen diesen beiden Routen.

7.1.3.2.13 ConfBlock

```
<confblock>    ::= <CONFLICTS> "{" {<confdef>} "}"
```

Vor dem ConfBlock steht das Wort `conflicts`, welches den Block identifiziert. Der gesamte Block von beliebig vielen Konfliktdefinitionen wird mit geschweiften Klammern umrahmt.

Der ConfBlock enthält die restlichen Informationen der *Route Conflict Table*, die nicht im PtConfBlock beschrieben werden – nämlich die weichen Konflikte. Treffen mindestens zwei aus verschiedenen Richtungen kommende Routen an einer Weiche aufeinander oder kreuzen sich zwei Routen, dann tritt ein sogenannter soft conflict an der Weiche bzw. Kreuzung auf. Analog zum PtConfBlock werden hier die in Konflikt stehenden Routen aufgeschrieben.

Die Angabe der Konflikte erfolgt wiederum doppelt, Erklärung dazu findet sich beim vorhergehenden Block.

Beispiel:

```
conflicts {
    route1 : route4, route2 ;
    route2 : route1, route3 ;
    route3 : route2, route4 , route6 ;
    route4 : route3, route1 , route5 ;
    route5 : route4 ;
    route6 : route3 ;
}
```

Dieses sind die weichen Konflikte der einzelnen Routen des oben genannten Beispieles in Abbildung 7.1 auf Seite 63. Beispielsweise hat die Route `route2` drei weiche Konflikte. Sie kreuzt sich, wie auf der Abbildung zu erkennen ist, mit der Route `route3` und mit der Route `route6` und anschließend trifft sie auf einer Weiche mit der Route `route4` zusammen. Nur anhand dieses Blockes kann man also nicht erkennen, ob es sich um eine Kreuzung oder ein Zusammentreffen handelt.

7.1.3.2.14 HWBlock

```
<HWBLOCK>       ::= <HARDWAREMAP> "{" {<HWDEFS>} "}"
<HWDEFS>        ::= <HWID> ":" <HWELID> {"," <HWELID} ";"
<HWELID>        ::= <SENSID> | <POINTID> | <SIGID> | <CROSSID>
```

Der HWBlock ordnet jeder im DefBlock deklarierten Hardwareklasse die ihr zugehörigen Elemente zu. Das den Block einleitende Schlüsselwort lautet `hardwaremap`, danach folgt in geschweiften Klammern eine beliebige Zahl von Zuordnungen. Jede einzelne Zuordnung beginnt mit dem Namen der Hardwareklasse gefolgt von einem Doppelpunkt, danach durch Kommata getrennt die einzelnen zugehörigen Elemente (mind. 1). Jede Zuordnung wird mit Semikolon abgeschlossen.

Zu beachten ist weiterhin, dass nur Sensoren, Weichen, Signale und einer Kreuzungsweiche angehörende Kreuzungen einer Hardwareklasse zugeordnet werden können, da dies diejenigen Elemente sind, über die der Steuerinterpreter die tatsächliche Steuerung vornehmen bzw. zur Laufzeit Informationen über den Zustand einholen kann.

Beispiel:

```
c1: g101, g102, g103, g104, g105, g201, g202, g203, g301,
    g302, g303, g304, g401, g402, g403, g404, g405, g406,
    g100, g200, g300, g400
c2: s2, s3, s1, s4, w2, w3, w1, w4;
```

(Ein sinnvolles Beispiel ist hier weniger gut möglich.)

7.1.3.3 Typregeln

Über die beiden Grammatiken hinaus gibt es ferner noch diverse Typregeln, die innerhalb eines kompletten Dokumentes erfüllt sein müssen, damit es der Sprachdefinition von TND genügt. Diese werden nun genannt:

- Im DefBlock darf jede PointID, SigID, SensID, MarkID, RouteID, CrossID oder HWID nur einmal vorkommen.

- Generell für alle Blöcke außer dem DefBlock: Jede PointID, SigID, SensID, MarkID, RouteID, CrossID oder HWID, die in einem der Blöcke vorkommt, muss auch im DefBlock vorkommen.
- Im CoordBlock muss jede PointID, SigID, SensID, MarkID oder CrossID, die im DefBlock definiert worden ist, genau einmal vorkommen.
- Im SigPropBlock muss jede SigID, die im DefBlock definiert worden ist, genau einmal vorkommen.
- Im SigPropBlock dürfen den SigIDs nur Wait- bzw. RR-Anzeigen zugeordnet werden (mittels Vorkommen der entsprechenden Token), deren Richtung das zugehörige Signal auch hat. (Bspw. darf ein sl-signal keine wait-right-Anzeige haben.)
- Im PointPropBlock muss jede PointID, die im DefBlock definiert worden ist, genau einmal vorkommen. Der PointPropBlock darf also nur entfallen, wenn im DefBlock keine PointID vorkommt.
- Im PointPropBlock darf einer PointID nur eine Fallback-Richtung zugeordnet werden (mittels Vorkommen des entsprechenden Tokens), deren Richtung die zugehörige Weiche auch hat.
- Im PointPropBlock dürfen einer PointID nicht gleichzeitig die Eigenschaften „breakable“ und „passive“ zugeordnet werden. (Die so beschaffene Weiche würde keinen Sinn machen - nicht stellbar, aber auch nicht von der stumpfen Seite her befahrbar.)
- Im SlipSwitchBlock darf keine CrossID oder PointID mehrfach vorkommen.
- Im RelBlock darf jedes Paar `<elementid-side>` nur einmal vorkommen.
- Im RelBlock darf jede MarkID nur einmal vorkommen.
- Im RelBlock darf innerhalb einer Relation nicht eine SensID mehrfach vorkommen.
- Im SigPosBlock muss jede SigID, die im DefBlock definiert worden ist, genau einmal vorkommen.
- Im RouteBlock muss jede RouteID, die im DefBlock definiert worden ist, genau einmal vorkommen.
- Im ClearBlock muss jede RouteID, die im DefBlock definiert worden ist, genau einmal vorkommen.
- Im HWBlock muss jede PointID, SigID oder SensID, die im DefBlock definiert worden ist, genau einmal vorkommen.

- Im HWBlock darf jede CrossID, die im DefBlock definiert worden ist, genau einmal vorkommen.

Diese Regeln sorgen im Wesentlichen dafür, dass TND-Dokumente mit einem Inhalt, der sinnlose oder sogar unmögliche Gleisnetze darstellen würde, so gut wie möglich verhindert werden - wenngleich an dieser Stelle nicht behauptet werden soll, dass nicht trotzdem noch die Notation eines sinnlosen Gleisnetzes möglich ist.

Silvia Graumann, Niklas Polke, Henrik Röhrup, Susanne Spiller, Arne Stahlbock

7.1.4 Schnittstellen

Es gibt mehrere Schnittstellen zu anderen Bestandteilen des Projekts – Daten werden sowohl in die TND-Form konvertiert als auch aus ihr ausgelesen. Da die TND sowohl Routeninformationen als auch Koordinaten und Beziehungen umfasst, wird eine vollständige Datei, die das gesamte Gleisnetz enthält und für die Verarbeitung im Projekt benötigt wird, aus verschiedenen Quellen zusammengesetzt:

- Die Informationen über das physikalisch vorhandene Gleisnetz wie Elementbeziehungen und Koordinaten werden vom CAD-TND-Konverter, der aus einer DXF-Datei die Informationen ausliest und in TND-Form in einer neuen Datei abspeichert, geliefert.

- Alle weiteren Informationen (z.B. die Daten der Verschlusstabellen) werden mittels des TND-Builders in die TND-Datei eingefügt.

- Prinzipiell kann auch jede Information von Hand eingefügt werden.

Eine TND-Datei wird von mehreren Arbeitsgruppen des Projekts weiterverarbeitet: *Simulator*, *HW/SW-Testgruppe*, *Modelchecker* und *Compiler*. Die Gruppen brauchen dabei nicht unbedingt alle Daten aus der TND:

- Der Simulator braucht aus der TND-Datei nur die Informationen, die nötig sind, um das Gleisnetz zu visualisieren. Ferner braucht er auch noch die Routeninformationen, um auf dem dynamischen Gleisnetz Straßenbahnen entlang vorher definierter Routen fahren zu lassen (s. Simulator, Kapitel 7.8 auf Seite 313).

- Der Steuerinterpreter, der vom Compiler die binären Projektierungsdaten bekommt, die letztlich aus der TND hervorgehen, benötigt keine Koordinaten, wohl aber alle Informationen aus den Verschlusstabellen und die Hardwaredaten.

- Modelchecker und HW/SW-Test als Verifikationsinstrumente für den Steuerinterpreter benötigen logischerweise dieselben Daten wie dieser, da ansonsten eine Verifikation fehleranfällig ist.

Silvia Graumann, Henrik Röhrup, Arne Stahlbock

7.1.5 Fortentwicklung der TND

Die soeben beschriebende Sprachdefinition ist bereits die dritte Version, die im Rahmen von TRACS entstanden ist. Die erste Version stand zum Ende des ersten Projektsemesters, musste aber im Laufe des dritten Semesters relativ umfassend überarbeitet werden, da sich aus einzelnen Gruppen neue Anforderungen bezüglich des durch die TND zu realisierenden Informationsumfanges ergaben. Eine weitere, jedoch nicht mehr so weitreichende Erneuerung wurde zu Beginn des vierten Semesters vorgenommen - dies ist die vorliegende Fassung. Was das Projekt TRACS betrifft, handelt es sich damit auch um die abschließende Version.

Arne Stahlbock

7.1.6 Reflexion

(Anmerkung: Da dieser Textabschnitt nicht direkt nach Abschluss der Arbeit in dieser Gruppe angefertigt wurde, sondern gut 1,5 Jahre danach, kann es sein, dass einige Umstände hier nicht mehr so umfassend behandelt werden, wie es bei einem früher entstandenen Text der Fall gewesen wäre.)

Die DSL-Gruppe arbeitete als solche nicht einmal ein ganzes Semester zusammen - sie wurde im Laufe des ersten Semesters eingerichtet und am Ende dessen auch wieder aufgelöst. Bezüglich ihrer Aufgabenerfüllung ist festzuhalten, dass am Ende des ersten Semesters auch eine zu diesem Zeitpunkt hinreichende Version der TND erstellt worden war, so dass ein weiterer Bedarf an einer kompletten Arbeitsgruppe für das Projekt nicht mehr bestand und die Auflösung somit zu Recht erfolgte, um die so freiwerdende Arbeitskraft an anderer Stelle einsetzen zu können. Die in der Folge dennoch gelegentlich nötigen Arbeiten (siehe auch den vorigen Abschnitt) konnten im Wesentlichen auch von einer Person erledigt werden, die zu diesem Zweck ihre eigentliche Tätigkeit unterbrach.

Probleme innerhalb der Gruppe (als es noch eine Gruppe war) existierten nicht. Bezüglich der Zusammenarbeit mit anderen Gruppen ist zu sagen, dass die DSL-Gruppe gerade im ersten Semester weitgehend unabhängig von anderen Gruppen arbeitete, was zusammen mit der Tatsache, dass zu diesem Zeitpunkt noch ein Weltmodell fehlte, dazu führte, dass die erste Version der TND nicht alle später noch aufkommenden Anforderungen erfüllte. Hier wären also andere Arbeiten (Erstellung des Weltmodells, Sammlung von Anforderungen an die zu entwickelnde Sprache) vor der Erstellung der TND nötig gewesen, die von der DSL-Gruppe – im Bezug auf das Weltmodell auch dem übrigen Projekt – zu diesem Zeitpunkt nicht erkannt worden waren.

Nach dem Aufkommen weiterer Anforderungen in den späteren Semestern konnte die TND jeweils relativ schnell auf einen neuen Stand gebracht werden. Hier wurden dann

auch die jeweiligen Forderungen der anderen Gruppen relativ klar kommuniziert, was ebenfalls im ersten Semester weitgehend ausblieb.

Insgesamt wird dennoch ein positives Fazit gezogen - die Aufgaben sind erfüllt worden, auch wenn eine andere Vorgehensweise möglicherweise schneller gewesen wäre.

Arne Stahlbock

7.2 Netzgraph und TND-Erzeugung

7.2.1 Überblick

Der Netzgraph ist die elektronische Beschreibung eines Gleisnetzes, d.h. er enthält eine vollständige Aufzählung aller vorhandenen Elemente (Sensoren, Signale, Weichen), die Nachbarschaftsbeziehungen zwischen Elementen (Gleisabschnitte) und letztendlich auch topologische Informationen (Koordinaten, Ausrichtungen). Der Netzgraph enthält nicht die Verschlusstabellen, also die „route definitions“, „route conflicts“, „point positions“ und „signal settings“. Die Informationen aus den Verschlusstabellen sind Bestandteil der TND, können aber nicht aus dem Netzgraphen gewonnen werden und müssen daher anderweitig in die TND eingefügt werden (siehe dazu Kapitel 7.3 auf Seite 123). Das selbe gilt für die Definition von Hardware-Klassen. Der Netzgraph zeigt als Zeichnung, wie die Gleisnetze mit ihren Steuerelementen aussehen, er kann aber auch in einem Dateiformat (DXF) gespeichert werden, welches als Text lesbar ist.

Der Netzgraph wird benötigt, um eine Repräsentation des kompletten, physisch vorhandenen Gleisnetzes zu erhalten, da dieses nicht zwangsläufig aus den Verschlusstabellen hervorgeht. In unserer Lösungsidee war es nun die Aufgabe, ein physikalisches Gleisnetz mittels eines CAD-Programmes in ein Dateiformat aufzunehmen und diese Datei wiederum in unsere TND-Darstellung umzuwandeln. Zur Validierung dieser Konvertierung wird die TND-Netzgraphbeschreibung mit Hilfe des Simulators (Abschnitt 7.8 auf Seite 313) visualisiert. Tritt ein Unterschied zum realen Gleisnetz auf, so ist die Konvertierung fehlerhaft.
Der so entstandene CAD-TND-Konverter ist der anspruchsvollste und komplexeste Bestandteil der Eingabetools für Gleisnetze im Projekt TRACS, da nur er die Möglichkeit hat, ein grafisch dargestelltes Gleisnetz in die benötigte TND-Darstellung umzuwandeln.

überarbeitet von Henrik Röhrup

7.2.2 CAD-System

7.2.2.1 Was ist CAD?

„CAD steht für Computer-Aided Design (Computerunterstütztes Zeichnen). Es ist wichtig, zwischen CAD-Applikationen und einfachen Zeichenprogrammen zu unterscheiden. Zeichnungen, die in einem CAD-System erstellt werden, repräsentieren Objekte aus der realen Welt in exakten originalen Abmessungen. Die wichtigsten Faktoren einer CAD-Zeichnung sind die Genauigkeit und die Ersichtlichkeit aller Details, die für die Produktion des entsprechenden Objektes notwendig sind.“ [Mus04]

7.2.2.2 CAD-Programm QCad

Wir benutzen die CAD-Software *QCad* (Version 2.x) für die Darstellung des Netzgraphen im Projekt TRACS.

„QCad ist ein 2D CAD System der Firma RibbonSoft (www.ribbonsoft.com). Das heisst, dass alles auf Ebenen projiziert dargestellt wird. Trotzdem können natürlich dreidimensionale Objekte dargestellt werden. Einige 2D Zeichnungen, die ein Objekt von verschiedenen Seiten darstellen, reichen in der Regel aus, um das Objekt vollständig in Form und Größe zu definieren.“[Mus04]
Dies ist für die Darstellung des Netzgraphen schon hinreichend.

QCad ist für die Betriebssysteme Linux, Unix, Windows^{TM}oder Mac OS X verfügbar.
„Außerdem ist QCad Teil der Open-Source Bewegung. Das bedeutet, dass Sie das Programm erweitern können, wenn Sie die nötigen Programmierkenntnisse haben.“[Mus04]

QCad ist wegen seiner Einfachheit für jedermann einsetzbar.
„QCad zielt auf Hobby-Anwender, gelegentliche Anwender und Leute, die keine CAD-Ausbildung haben.“[Mus04]
Dieses Anwenderprofil ist für TRACS durchaus zutreffend.

Zur Bedienung:
„Die meisten Funktionen von QCad können über das Menü erreicht werden. Für CAD-spezifische Funktionen kann es bequemer sein, die Werkzeugleiste links im Hauptfenster zu verwenden. Die Werkzeugleiste zeigt jeweils diejenigen Funktionen an, welche zur Zeit am meisten Sinn machen.“ [Mus04]

Das Format der Ausgabedatei von QCad ist DXF. Dieses Format ist auch unter AutoCAD (CAD-Software der Firma Autodesk) verfügbar. Wegen der durchaus hohen Verbreitung von AutoCAD sollten Kompatibilitätsprobleme mit diesem Format minimiert sein.

7.2.2.3 DXF-Format

„Das DXF-Format ist eine etikettierte Datendarstellung aller Informationen, die in einer CAD-Zeichung enthalten sind. Praktisch können alle spezifischen Informationen in einer Zeichnungsdatei im DXF-Format dargestellt werden.“ (Übersetzt aus [Aut03])

Genauer gesagt gibt es immer Zeilenpaare, von denen die erste Zeile einen ein- bis dreistelligen Zahlencode enthält, welcher im globalen Zusammenhang einen bestimmten Parameter darstellt. In der zweiten Zeile ist der Wert dieses Parameters vermerkt. Abhängig von der Art des Zeichenelements kann der Code einmal den einen und ein

anderes Mal einen anderen Zweck erfüllen.

Eine DXF-Datei ist in mehrere Sektionen (Abschnitte) aufgeteilt. Jede Sektion erfüllt ihren eigenen Zweck. So gibt es zum Beispiel die Sektion **ENTITIES**, in der die einzelnen Elemente (oder ein Verweis auf eine Gruppe von Elementen (Block)) einer Zeichnung definiert sind. Die Sektion **BLOCKS** beinhaltet eben solche Gruppen von Elementen, sogenannte Blöcke. Die Sektionen selbst sind ebenfalls untergliedert, wobei die Anzahl dieser Unterteilungen von der Anzahl der Elemente (für die Sektion **ENTITIES**) oder Blöcke (**BLOCKS**) abhängt. In jeder dieser Unterteilungen sind alle zusammengehörigen Code-Wert-Zeilenpaare enthalten. Die genaue Reihenfolge ist nicht weiter definiert. Ebenso müssen nicht alle Codes, welche z.B. zu einem Zeichenelement gehören, enthalten sein, da sie möglicherweise optional oder mit Standard-Werten vorbelegt sind.
Darüberhinaus gibt es noch die Sektionen **HEADER** und **TABLES**.

Genauere Informationen sind der DXF-Referenz [Aut03] zu entnehmen.

Die DXF-Dateien können ohne CAD-Programm mit normalem Texteditor geöffnet werden, und der Inhalt der Dateien ist lesbar. Das bedeutet, man kann relativ einfach die Informationen aus den DXF-Dateien bekommen, egal ob ein CAD-Progamm verfügbar ist oder nicht. Es ist allerdings die angesprochene DXF-Referenz notwendig, um das Format als solches verstehen zu können.

stark überarbeitete Fassung: Henrik Röhrup

7.2.3 Arbeitspakete

Die Arbeitspakete der Netzgraph-Gruppe sind in zwei wesentliche Bereiche einzuteilen:

- CAD-System zur Aufnahme eines Gleisnetzes
- Konverter-Software zur Generierung der Netzgraph-Informationen eines Gleisnetzes und Erstellung einer TND-Datei

7.2.3.1 Einarbeitung in CAD-System

Das erste eigentliche Arbeitspaket war die Einarbeitung in das CAD-System QCad, um dieses sinnvoll zum Zeichnen der zu konvertierenden Gleisnetze einsetzen zu können. Das Hauptaugenmerk lag auf dem Erlernen von Techniken, mit deren Hilfe man ohne größeren Aufwand Netzpläne von Bahnsystemen erstellen kann.

7.2.3.1.1 QCad und die Objekt-Bibliothek QCad bietet die Möglichkeit, sog. Blöcke zu definieren, welche dann innerhalb einer Zeichnung mehrfach Anwendung finden können (im Prinzip also Ausschnittszeichnungen häufig verwendeter Zeichen-Elemente). Aus diesem Grund wurde eine Objekt-Bibliothek erstellt, welche Blöcke für alle in einem Gleisnetz vorkommenden Elemente, also alle Typen von Weichen, Signalen, Sensoren, etc. enthält. Durch Einsatz dieser Bibliothek ist es möglich, innerhalb des Dateiformates (DXF) diese Elemente wiederzufinden, wodurch das spätere Einlesen dieser Datei erst auf sinnvolle Weise ermöglicht wird.

Die Bibliothek ist in mehrere Kategorien unterteilt: **points** (sämtliche Weichen), **sensors** (sämtliche Sensoren), **signals** (sämtliche Signale), **track** (normale Kreuzungen) und **others** (Eintritts- und Ausgangspunkte, Markierungen, etc.).

Diese Objektbibliothek muss vom Anwender in das entsprechende Verzeichnis kopiert werden, bzw. der Anwender muss in den Programm-Einstellungen der CAD-Software den Pfad zu dieser Bibliothek als Quelle für DXF-Bibliotheken hinzufügen (s.a. Abschnitt 7.2.3.4). Ob diese Bibliothek auch mit anderen zum DXF-Format kompatiblen CAD-Programmen problemlos genutzt werden kann, ist nicht klar, da der Netzgraphgruppe nur das Programm QCad zur Verfügung stand. Wie diese Einbindung in QCad erfolgen kann, wird im Handbuch für die Netzgraph-Konverter-Tools beschrieben.
Abbildung 7.2 zeigt ein kleines Beispiel für ein mit der DXF-Objekt-Bibliothek ge-

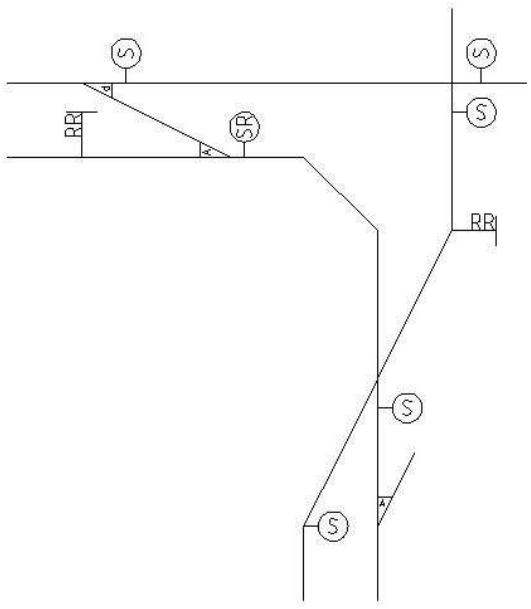


Abbildung 7.2: Beispiel eines mittels der QCAD-Objekt-Bibliothek erstellten Teilgleisplans

zeichnetes Gleisnetz. (Anmerkung: Die für dieses und die weiteren Beispiele verwendete Objekt-Bibliothek ist mittlerweile veraltet. Die Unterschiede sind allerdings nur marginal, so dass hier keine neuen Grafiken angefertigt wurden.)

Um die Übersichtlichkeit und Darstellbarkeit von großen Gleisnetzen in der CAD-Software zu verbessern, haben wir uns dazu entschieden, die verschiedenen Arten von Elementen auf verschiedenen Ebenen (Layern) darzustellen, welche bei Bedarf ein- und ausgeblendet werden können. Es ist also möglich nur die Gleisansicht zu betrachten und bei Bedarf Sensoren und Signale ein- oder auszublenden. Dies erleichtert unserer Ansicht nach die Erstellung von solchen Gleisplänen, da so z.B. ein schnellerer Überblick über das Schienennetz erlangt werden kann. Für die direkte Weiterverarbeitung im anschließend beschriebenen CAD-TND-Konverter sind die Layer aber nicht notwendig.

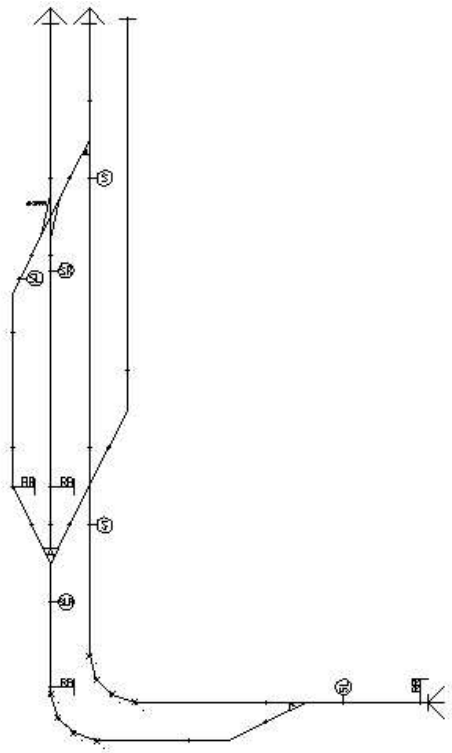


Abbildung 7.3: Gleisnetzzeichnung mit allen Layern

Die Abbildung 7.3 zeigt eine Gleisnetzzeichnung mit allen Layern, während bei Abbildung 7.4 auf der nächsten Seite die Sensoren und einige andere Elemente ausgeblendet sind. Abbildung 7.5 auf Seite 88 zeigt hingegen nur die Gleise selbst.

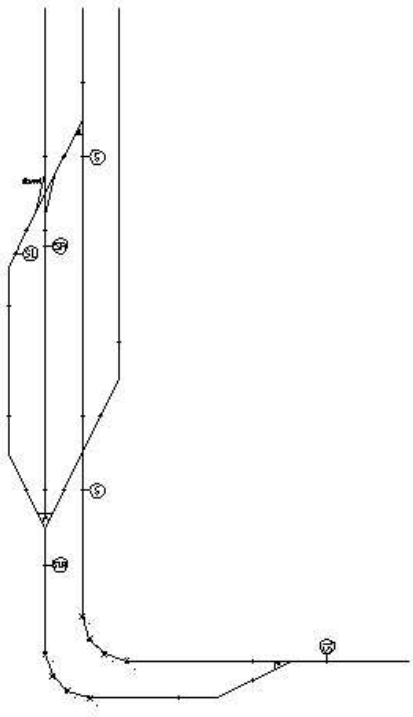


Abbildung 7.4: Gleisnetzzeichnung ohne Sensoren und Ein-/Ausgänge

7.2.3.1.2 DXF-Dateiformat Im Laufe des dritten Projekt-Semesters wurden noch einige Erweiterungen, bzw. Modifikationen an der Bibliothek vorgenommen. So wurden zum Beispiel im Rahmen der Überarbeitung des Weltmodells Kreuzungsweichen hinzugefügt. Auch fehlten bisher Einfahrts- bzw. Ausfahrtspunkte. Zur Erleichterung des Erkennens von Nachbarschaftsbeziehungen wurden die Übergangspunkte (Ein- und Ausgänge von z.B. Weichen) entsprechend markiert, da es sonst sehr aufwändig gewesen wäre, diese innerhalb eines Element-Blocks zu bestimmen. Dadurch wurde auch die weitere Arbeit im Vergleich zur ersten Idee stark vereinfacht, da nur noch vergleichsweise wenig Informationen aus der DXF-Datei ausgelesen werden müssen.

Bei der Untersuchung des uns anfangs zur Verfügung stehenden Programms in der Version 1.5.x stellten wir fest, dass es für unsere Zwecke unzureichend ist, da die vorliegende Version zwar Blöcke einlesen und darstellen und somit auch mit unserer Objekt-Bibliothek umgehen, aber intern diese Elemente nicht mehr als solche Blöcke abspeichern kann, sondern stattdessen diese in die einzelnen Bestandteile eines Blocks (wie z.B Linien, Kreise, etc.) auflöst, was das spätere korrekte Einlesen zum Zwecke der Konvertierung nahezu unmöglich machen würde. Daher haben wir uns dazu entschieden, die aktuelle Version 2.x zu verwenden, da diese die Unzulänglichkeiten der Vorgängerversion

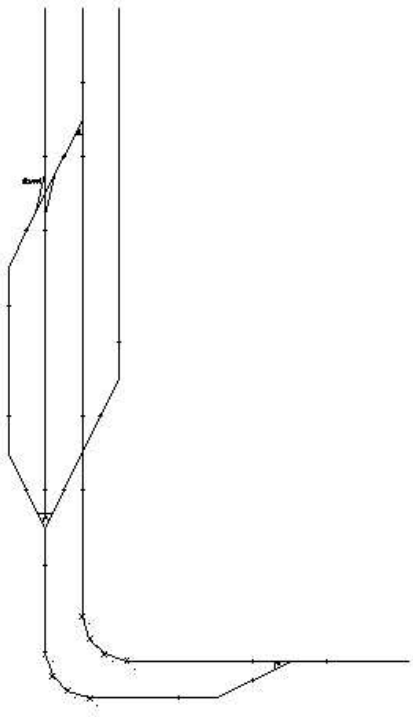


Abbildung 7.5: Gleisnetzzeichnung mit ausschließlich Gleisen

nicht aufweist. Der angenehme Nebeneffekt war auch der weitaus größere Funktionsumfang, welcher bequemere Nutzungsmöglichkeiten, wie z.B. deutlich umfangreichere Menüstrukturen, klarer formulierte Dialoge, etc., bereitstellte.

Zur Einarbeitung gehörte auch die Untersuchung des Ausgabeformates (DXF), da die DXF-Datei letztlich die Eingabedatei für die nächste Aufgabe ist.
Zum DXF-Format ist folgendes anzumerken: Es existieren z.Z. rund zwei Dutzend Versionen des DXF-Formates, da dieses Dateiformat von der selben Firma (Autodesk) entwickelt worden ist, welche auch die Software AutoCAD vertreibt und bei jeder nahezu jährlich erscheinenden neuen Version dieser Software erscheint auch eine überarbeitete Version des DXF-Dateiformates.
Bei der Untersuchung des uns vorliegenden DXF-Formates hat sich herauskristallisiert, dass eigentlich nur zwei Sektionen von wirklichem Interesse sind. Dies ist zum einen die Sektion **ENTITIES**, wo letztlich alle Zeichenelemente (z.B. Linien, Punkte, etc.) mit ihrer genauen Lage, Ausrichtung und sonstiger Formatierung vermerkt sind. Zum anderen ist dies die Sektion **BLOCKS**, in der die zuvor genannten Blöcke definiert sind, welche dann in der Sektion **ENTITIES** nur noch als **INSERT** (Einfügung) wie ein normales Zeichenelement vermerkt werden.

Ein Auszug aus einer Beispiel-DXF-Datei, sowie die für uns relevanten DXF-Elemente sind im Anhang (B.3 auf Seite 463) zu finden.

7.2.3.2 CAD-TND-Konverter

Um ein per CAD-System graphisch erstelltes Gleisnetz weiterverarbeiten zu können, wird ein Konverter benötigt, der das Dateiformat der CAD-Software in unser TND-Format transformiert. Dies ist das zweite, wichtigste und zugleich aufwändigste Arbeitspaket im Teilprojekt Netzgraph. Die Vorarbeit - nämlich das Kennenlernen des Ausgabeformates der CAD-Software - wurde im vorangegangenen Arbeitspaket durchgeführt. Weiterhin ist zu beachten, dass der Konverter von der Definition des TND-Formats abhängt.
Um so einen Konverter zu implementieren gibt es mehrere Wege und mehrere verwendbare Programmiersprachen.

7.2.3.2.1 Entscheidung für Java™ als Programmiersprache Wir haben uns für Java™ als Programmiersprache für den CAD-RND-Konverter entschieden. Der Hauptgrund für diese Entscheidung war die Tatsache, dass zu jenem Zeitpunkt eben diese Sprache von allen Mitgliedern der Netzgraph-Gruppe am besten beherrscht und somit erwartet wurde, dass die Implementierung in dieser Sprache am effektivsten voranschreiten würde. Da während der Implementierung diesbezüglich auch keine größeren Probleme aufgetreten sind, erachten wir diese Wahl im Nachhinein als richtige Entscheidung.
C- oder C++-Kenntnisse waren zwar vorhanden, aber die Erfahrungen beschränkten sich zu jenem Zeitpunkt auf ein Minimum, so dass eine Verwendung dieser Sprachen einen sehr hohen Einarbeitungsaufwand erfordert hätte. Ein Vorteil von C/C++ gegenüber Java™war zu jenem Zeitpunkt nicht erkennbar und ist es auch am Ende der Arbeit am Konverter nicht.
Java™ist eine objektorientierte Programmiersprache (OOP). Diese Eigenschaft machen wir uns beim Auslesen der DXF-Datei und auch beim Erzeugen der TND-Datei zu Nutze.
Da das Konvertieren einer Datei in ein anderes Format keine zeitkritische Angelegenheit ist, ist es auch kein Problem Java™, was im Vergleich zu C++ bei der Ausführung der Programme recht langsam ist, zu verwenden.

7.2.3.2.2 Entscheidung gegen Übersetzerbauwerkzeuge Im Verlaufe der ersten Semesterhälfte des dritten Projekt-Semesters wurde außerdem endgültig die Entscheidung getroffen, beim Einlesen der DXF-Daten auf die Verwendung von Übersetzerbauwerkzeugen zu verzichten - auch wenn die Aufgabe mit solchen in der Praxis erprobten Werkzeugen theoretisch realisierbar wäre.

Bei der Entscheidung wurde hauptsächlich das Werkzeug *CUP Parser Generator for Java™* betrachtet, welches vom Simulator (Abschnitt 7.8 auf Seite 313) verwendet wird. Es wurde der Versuch unternommen, sich in dieses Werkzeug so einzuarbeiten, so dass es auch in der Gruppe Netzgraph Anwendung finden kann. Die Einarbeitungszeit stellte sich dabei aber schnell als recht groß heraus, da zu jenem Zeitpunkt nur noch eine Person am gesamten Netzgraph-Teil-Projekt arbeitete. Auch ergab sich augenscheinlich kein überzeugender Vorteil - weder im Hinblick auf eine Beschleunigung der Implementierung, noch eine Vereinfachung der Datengewinnung - , so dass die Idee der weiteren Verwendung verworfen wurde.

Wie bereits beschrieben, ist eine DXF-Datei in mehrere Teile untergliedert und darin ggf. in mehrere als Einheit anzusehende Untergruppierungen. Innerhalb dieser Untergruppierungen sollten für den jeweiligen Zweck (z.B. ein Textfeld) bestimmte DXF-Codes und zugehörige Werte vorhanden sein. Nach weitergehenden Tests während der Einarbeitungsphase und des dritten Semesters (Überarbeitung der DXF-Objekt-Bibliothek) stellte sich heraus, dass nicht zwangsläufig alle Codes vorkommen müssen, um in der CAD-Software eine korrekte Darstellung zu gewährleisten - jedenfalls gilt dies für das verwendete CAD-System.
Dazu gesellte sich auch noch die Tatsache, dass in neueren Versionen des DXF-Formats zahlreiche Zusatzinformationen (hauptsächlich zur Formatierung einzelner Elemente (Farbcodierungen, Linientypen, etc.) und weiterer nicht näher untersuchte Informationen) vorkommen, welche bei uns absolut keine Verwendung finden. Die Menge an Informationen hat sich innerhalb weniger Versionsschritte vervielfacht, so dass das aktuelle DXF-Format insgesamt letzlich etwas undurchschaubar und verworren anmutet.
Im Zuge der Überarbeitungs-Phase wurde auch versucht, eine mehr oder weniger exakte DXF-Grammatik aufzustellen, was sich aber aufgrund der zuvor genannten Eigenschaften des DXF-Formats und der hohen Anzahl an verschiedenen Versionen nicht gerade als einfach und eher als kaum realisierbar herausgestellt hat. Daher ist die im Anhang dieses Berichts enthaltene DXF-Grammatik (siehe Anhang B.2 auf Seite 461) vereinfacht und beschränkt sich nur auf die für den Netzgraphen direkt relevanten Sektionen.

Wie erwähnt, stellte sich ebenfalls im Laufe des dritten Projekt-Semesters heraus, dass von den insgesamt für die grafische Darstellung relevanten Informationen der DXF-Datei nur ein geringer Teil für die interne Weiterverarbeitung relevant ist (Koordinaten (sowohl die des gesamten Elements, als auch die der Berührungspunkte mit anderen Elementen) und der Name eines Elements aus der erstellten DXF-Bibliothek), sowie ggf. das eine oder andere Textfeld). Der Rest ist zu vernachlässigen.
Dies stellte sich leider erst so spät heraus, da die ursprünglichen Überlegungen noch ein anderes Arbeitspaket (den TND-DXF-Rück-Konverter, Abschnitt 7.2.3.3 auf Seite 103), welches in der Zwischenzeit entfallen ist, mit einbezogen. Bei diesem Paket, wo es um die Erstellung einer DXF-Datei gegangen wäre, wären andere Informationen von Bedeutung

gewesen, um die Korrektheit einer DXF-Datei zu gewährleisten.

Die aus den DXF-Daten zu gewinnenden Informationen stellen also nur als einen Bruchteil des Datenumfangs dar, welche eine DXF-Datei eigentlich zu bieten hat. Der Aufwand zum Gewinnen der detaillierten Informationen erscheint daher mit der gewählten Methode auch mit geringerem Aufwand machbar.
Der Einsatz von Tools wie *CUP Parser Generator for Java*^{TM} ist für wirklich große Projekte sicher sehr sinnvoll. Ob der Einsatz für diese Aufgabe sinnvoll gewesen wäre, ist eine Frage, über welche man sicher streiten kann. Nach Betrachtung der oben beschrieben Eigenschaften erschien uns dies hier nicht unbedingt sinnvoll zu sein, da der Zeitaufwand enorm gewesen wäre.
Es macht also letztlich wenig Sinn, sich mit der Einarbeitung in derartige Werkzeuge zu beschäftigen, wenn diese einen größeren Zeitaufwand erfordert, als der aus einer eventuellen Anwendung des Werkzeugs resultierender Zeitgewinn ist, zumal es gar nicht sicher ist, dass sich in diesem Falle wirklich ein nenneswerter Zeitgewinn ergeben hätte.

Aufgrund der bereits mehrfach angesprochenden Versions-Vielfalt des DXF-Formats garantiert die Netzgraphgruppe eine problemlose Verarbeitung auch nur für die zum Zeitpunkt der Implementierung des Konverters von QCad angebotenenen Versionen (R12 und AutoCad2004). An diesen Versionen sind keine Änderungen zu erwarten, so dass das Argument für Übersetzerbau-Werkzeuge hinsichtlich „Grammatik“-Änderungen und einem Mehraufwand beim Verzicht auf diese Werkzeuge aus Sicht des Netzgraphen zu vernachlässigen ist.
Das Argument der geringeren Fehleranfälligkeit bei jenen Werkzeugen kann man unter Umständen weiterhin anführen, sofern man die Versionsvielfalt mit diesen Werkzeugen in den Griff bekäme, jedoch sah die Netzgraphgruppe aufgrund der Art und Menge der einzulesenden Daten kein wirklich großes Fehlerpotenzial, welches nicht mit geringem Aufwand in den Griff zu bekommen gewesen wäre.
Außerdem bedeutet ein Verzicht auf derartige Werkzeuge noch lange nicht, dass das Ergebnis am Ende ein schlechteres sein wird...

7.2.3.2.3 Klassendiagramm Die Implementierung dieses Konverters basiert auf einer Überlegung, in welcher es im Groben drei Haupt-Klassen und einige Neben-Klassen gibt, die im folgenden Abschnitt beschrieben werden sollen.

Das Klassendiagramm (Abbildung 7.6) zeigt die erstellten Klassen des Netzgraph-Konverter-Paketes. Die Klassen *ConfigReader* (Abbildung 7.7), *NGElement* (Abbildung 7.11), *NGElementList* (Abbildung 7.9), *NGLines* (Abbildung 7.10) und *NGSystem* (Abbildung 7.14) sind Hilfsklassen. Die übrigen Klassen sind die Hauptklassen des Konverters. Die Klasse *TracsDXF2TND* (Abbildung 7.13) stellt letzten Endes das eigentliche Programm dar, welches die Konvertierung als Ganzes durchführt. Die Klassen *DXFSe-*

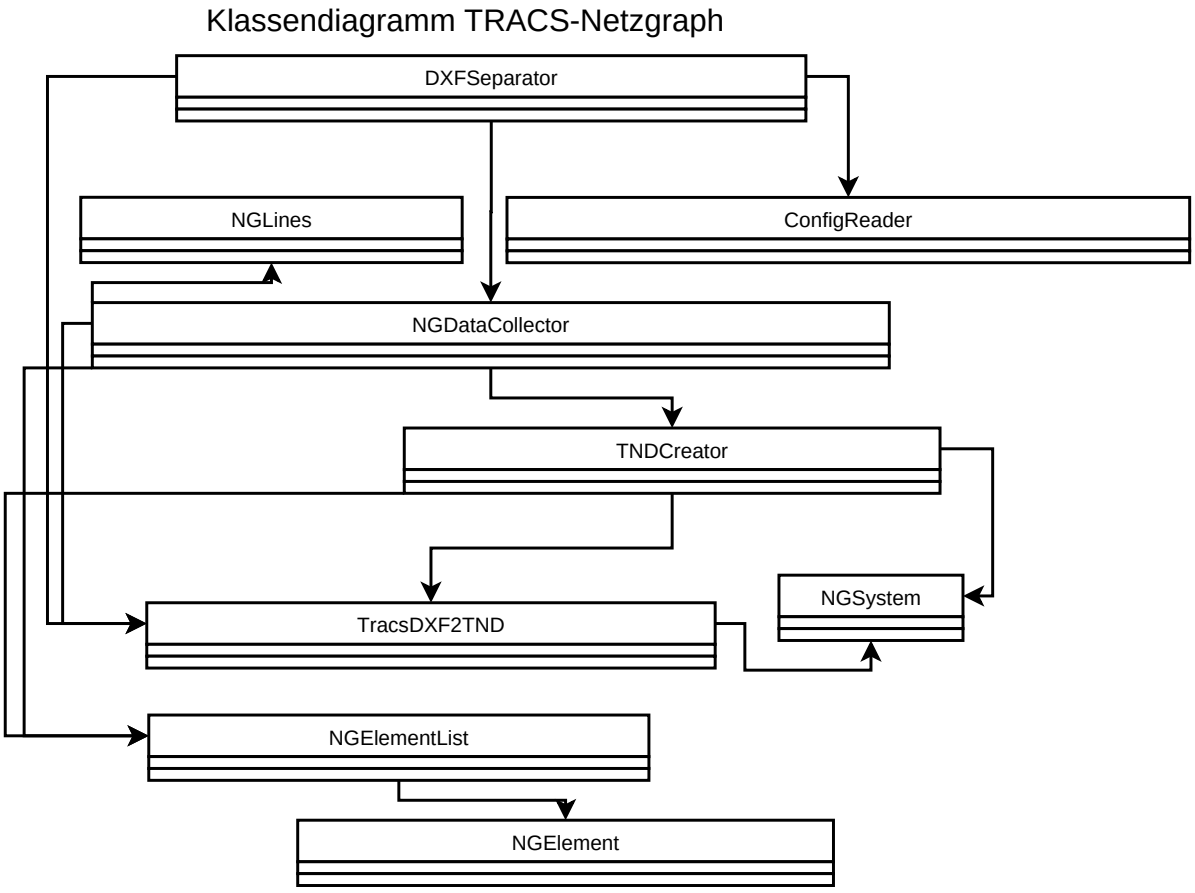


Abbildung 7.6: vereinfachte Darstellung des Klassendiagramms für den CAD-TND-Konverter

parator (Abbildung 7.7), *NGDataCollector* (Abbildung 7.8) und *TNDCreator* (Abbildung 7.12) werden von der zuvor genannten Klasse nacheinander aufgerufen. Sie sind aber auch als eigenständige Programme nutzbar, so dass mit deren Hilfe einzelne Informationen anzeigbar sind, bzw. Teilschritte einzeln durchgeführt werden können, was insbesondere für das Debugging Sinn ergibt. Daher haben auch alle drei Klassen eine eigene Main-Methode.
Die Klasse *ConfigReader* wird von der Klassen *DXFSeparator* aufgerufen, um ggf. Initialisierungsdaten einzulesen. Ursprünglich sollte sie auch von den übrigen Hauptklassen aufgerufen werden, jedoch ergab sich hier letzlich kein wirklicher Anwendungszweck, so dass diese Idee wieder verworfen wurde. Die Klassen *NGElement*, *NGLines* und *NGElementList* sind für die Handhabung von Objekten zuständig, welche von *NGDataCollector* erzeugt und von *TNDCreator* ausgelesen und weiterverabeitet werden. Daher sind sämtliche Methoden als **protected** (innerhalb des Paketes öffentlich) und die Attribute

als **private** (nicht öffentlich) markiert.
Die Pfeile zeigen, welche Klasse von welcher anderen Klasse aufgerufen / instanziiert wird (Pfeil von A nach B: Klasse A greift auf Klasse B zu).

7.2.3.2.4 DXFSeparator.java Die Grundüberlegung für diesen Konverter war es, vor jeglicher weitergehenden Verarbeitung der DXF-Datei erst einmal die relevanten Informationen aus dieser Datei herauszufiltern und den Rest auszusortieren um ein gewisses Debugging zu ermöglichen, bzw. zu vereinfachen. Dazu wurde eine Klasse (*DXF-Separator.java*) implementiert, welche die DXF-Datei komplett Zeile für Zeile einliest und dabei die weiter oben beschriebenen Sektionen **ENTITIES** und **BLOCKS** ausliest und in separate Dateien abspeichert (*blocks-tmp.tng* und *entities-tmp.tng*). Dabei werden außerdem alle für uns irrelevanten Informationen verworfen. Die Codes für relevante Informationen sind in einer externen Konfigurationsdatei definiert, welche mit Hilfe einer Hilfsklasse (*ConfigReader.java*, Seite 94) eingelesen wird.
Die Realisierung über zwei getrennte, temporäre Dateien wurde auch bis zum Ende beibehalten, da so auch für den Endnutzer eine gewisse Debugging-Möglichkeit besteht, weil diese Klasse auch als eigenständiges Programm ausgeführt werden kann - neben dem eigentlichen Aufruf durch das Hauptprogramm (*TracsDXF2ND.java*, Seite 102).
Das detaillierte Klassendiagramm (Abb. 7.7) zeigt die in dieser Klasse verwendeten Methoden und Attribute.

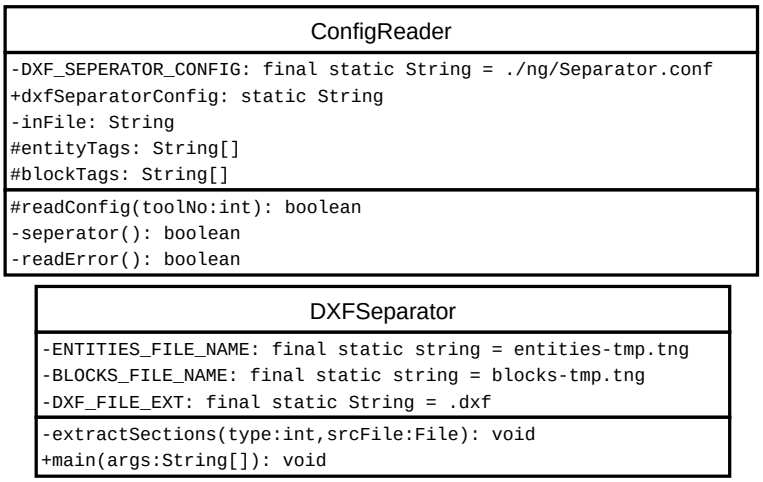


Abbildung 7.7: Detailansicht Klassendiagramm DXFSeparator und ConfigReader

Im Wesentlichen gibt es hier neben der Main-Methode nur eine Methode (*extractSections()*), welche den Vorgang der Extrahierung ausführt.

Dieser erste Teil des DXF-TND-Konverters wurde gegen Ende des 2. Projektsemesters weitestgehend fertiggestellt und seitdem nur marginalen Änderungen unterworfen. Die wesentlichen Tests wurden am Anfang des dritten Projekt-Semesters durchgeführt und nach erfolgtem Bugfixing funktionierte diese Klasse wie gewünscht. Im Anhang (siehe B.3.1 auf Seite 465) befindet sich eine (gekürzte) DXF-Beispiel-Datei mit Ergebnis nach Anwendung der DXFSeparator-Klasse (siehe B.3.2 auf Seite 465 und B.3.3 auf Seite 467).

7.2.3.2.5 ConfigReader.java Die Klasse *ConfigReader.java* ist eine Hilfsklasse. Sie wurde zum Einlesen von Konfigurationsdaten entwickelt und war hauptsächlich für die Klasse *DXFSeparator.java* gedacht, ist aber darauf ausgelegt auch für weitere Klassen Konfigurationsdateien einzulesen, was aber im Laufe des Projekts nicht weiter verfolgt wurde, da für die übrigen Klassen nicht der zwingende Bedarf bestand. Ein Beispiel einer Konfigurationsdatei mit Formaterklärung befindet sich im Anhang (siehe D.1 auf Seite 473).
Das detaillierte Klassendiagramm (7.7 auf der vorherigen Seite zeigt u.a. die in dieser Klasse verwendeten Methoden und Attribute.

7.2.3.2.6 NGDataCollector.java Die nächste wichtige Haupt-Klasse (*NGDataCollector.java*) in diesem Lösungsansatz ist für das eigentliche Einlesen der DXF-Informationen zuständig und bildet einen zentralen Bestandteil des Konverters. Diese Klasse wird durch das Hauptprogramm (*TracsDXF2TND.java*, Seite 102) und auch von der dritten Hauptklasse des Konverters (*TNDCreator.java*, Seite 98) aufgerufen, kann aber auch direkt aufgerufen werden (für Debugging). Die Implementierung dieser Klasse zog sich bis ins letzte Projektsemester hin, da zwischenzeitlich einige unerwartete Probleme auftraten und auch einige Details hinsichtlich der Nachbarschaftsbeziehungen nicht wirklich bedacht wurden. Es werden nun aber alle in einem DXF-Gleisnetz vorkommenden Elemente erkannt und in entsprechenden Java-Objekten abgespeichert.

Dabei werden die Koordinaten und die Art, sowie der Name des Elements (z.B. eine Links-Rechts-Weiche mit dem Namen „w359“) und evtl. vorhandene Eigenschaften ausgelesen. Dazu kommen die benötigten Informationen zu Nachbarschaftsbeziehungen, welche aus Berührungspunkten, bzw. Verbindungslinien (Gleisgeraden) abgeleitet werden. Dieser zuletzt genannte Teil hatte zwischenzeitlich einige Fehler verursacht, welche nur mit mühsamer Arbeit langsam ausgefiltert werden konnten.
Ein weiteres Problem ergab die Tatsache, dass Objekte in einer Zeichnung auch zu ihrer relativen Ausrichtung rotiert dargestellt werden können. Durch diese Rotation veränderten sich logischerweise auch die absoluten Koordinaten der Berührungspunkte mit anderen Elementen. Dieser Umstand wurde aufgrund der anfänglichen Planung ohne derartige Berührungspunkte (es sollten ursprünglich die einzelnen Gleisnetzelemente

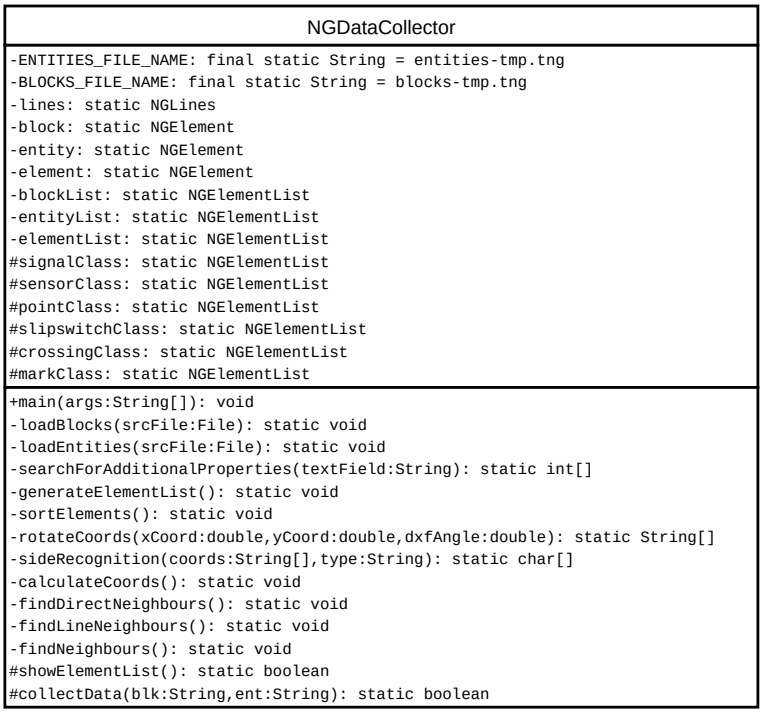


Abbildung 7.8: Detailansicht Klassendiagramm NGDataCollector

(Teilzeichnungen / Blöcke) selbst zerlegt und daraus die Zusammenhänge herausgefiltert werden) nicht vollkommen bedacht. Dazu kam noch, dass bei der Neuberechnung zwangsläufig einige Rundungsfehler auftraten und diese erst beseitigt werden mussten. Die Daten werden nach Tests mit DXF-Zeichnungen kleinerer Testgleisnetze ordnungsgemäß ausgelesen, sofern die Gleisnetzzeichung den im Handbuch (7.2.3.4 auf Seite 104) beschriebenen Regeln entspricht und der Zeichner keine Fehler produziert hat. Zeichenfehler sind aber kaum abfangbar, da die QCAD-Software nicht von TRACS geliefert wird, die Eingabe (oder vielmehr das Abspeichern) aber der sinnvollste Zeitpunkt für entsprechende Fehlerchecks wäre. (Zu diesem Umstand folgt noch eine allgemeine Anmerkung mit einer anderen Lösungsidee (7.2.5.8 auf Seite 120).)
Die so gewonnen Informationen werden nun von einer weiteren Klasse des Konverters (*TNDCreator.java*, Seite 98) weiterverarbeitet. Dazu werden die erzeugten Objekte in ArrayLists (einem Objekt-Typ in Java™, in welchem mehrere andere Objekte abspeicherbar sind) einsortiert, welche als eine Art Sortiercontainer dienen. Diese ArrayLists oder besser gesagt Objekte vom Typ NGElementList (vgl. 7.2.3.2.8 auf Seite 97) werden später von der Klasse *TNDCreator.java* entprechend ausgelesen.
Die Detailansicht des Klassendiagramms für die hier beschriebende Klasse (siehe Abbildung 7.8) zeigt die Liste der verwendeten Attribute und Methoden. Hier werden unter anderem die angesprichenden Sortier-Container defininiert, in welche letztlich alle Gleis-

netzelemente durch die Methode *sortElements()* einsortiert werden.
Die Methoden *loadBlocks()* und *loadEntities()* lesen die durch den *DXFSeparator* erzeugten Zwischendateien *entities-tmp.tng* und *blocks-tmp.tng* ein und sammeln die Informationen zu einzelnen Zeichnungsobjekten in Objekte vom Typ *NGElement* (siehe dazu 7.2.3.2.7).
Da sich in einer DXF-Datei sowohl Zeichnungsblöcke befinden, welche ein Gleisnetzelement enthalten, als auch solche, die nur zur Gruppierung von Gleisnetzelementen und deren Benennung (ein einfaches Textfeld) zuständig sind, müssen erstere noch herausgefiltert werden, damit die eigentliche Weiterverarbeitung der Informationen aus einer DXF-Datei fortgesetzt werden kann. Hierfür ist die Methode *generateElementList()* zuständig.
Weitere Methoden sind für die Neuberechnung von absoluten Koordinaten zuständig. Dies ist zum Beispiel bei Elementen notwendig, die zu ihrer Normalausrichtung gedreht sind, da intern jedes Element als solches nur relative Koordinaten liefert.
Manche Element-Typen besitzen außerdem bestimmte Eigenschaften (Properties), welche erkannt werden müssen. Hierbei handelt es sich um Weichen- und Signal-Elemente. Auch hierfür gibt es eine eigene Methode (*searchForAdditionalProperties()*).
Zur Erkennung von Nachbarschaftsbeziehungen zwischen einzelnen Elementen ist es notwendig, die Seiten (oder Ein- und Austrittspunkte) der Elemente zu erkennen, da diese in der TND-Datei von enormer Bedeutung sind. Dies wird mit der Methode *sideRecognition()* erledigt. Die Nachbarschaftsbeziehungen selbst werden letzlich in zwei Schritten gesucht:

- direkte Nachbarschaften: zwei Elemente treffen sich in ihren definierten Ein- / Austrittspunkten
- indirekte Nachbarschaften: zwei Elemente sind über Linien (einfache Schienenstücke) miteinander verbunden

Die Methode *findNeighbours()* vereint diese beiden Schritte in einem Aufruf.
Zu dieser Klasse ist noch zu sagen, dass Slipswitches - also Kreuzungsweichen - in der stabilen Version nicht unterstützt werden, da diese Funktionalität nicht mehr ausreichend getestet werden konnte. In der Test-Version ist die Funktion aber mit Einschränkungen vorhanden. (siehe dazu auch Abschnitt 7.2.3.2.13 auf Seite 103)

7.2.3.2.7 NGElement.java Zum Erzeugen der zuvor beschriebenen Objekte wird eine weitere Hilfsklasse (*NGElement.java*) benutzt. In einem solchen NGElement-Objekt werden alle für ein Gleisnetzelement relevanten Daten abgespeichert (Name, Typ, Koordinaten, Nachbar-Elemente, sowie weitere Informationen, um das Element im Gesamtkontext genau zu spezifizieren).
Diese Objekte werden sowohl für Elemente (Entities), als auch ganze Blöcke aus der

DXF-Datei genutzt, aber auch als Java™-Objekt für die später in die TND zu schreibenden Gleisnetzelemente selbst.
Diese Klasse ist sozusagen eine Art „Universalklasse“.
Die Ansicht des Klassendiagramms (Abb. 7.11 auf Seite 99) zeigt die in dieser Klasse verwendeten Methoden und Attribute.

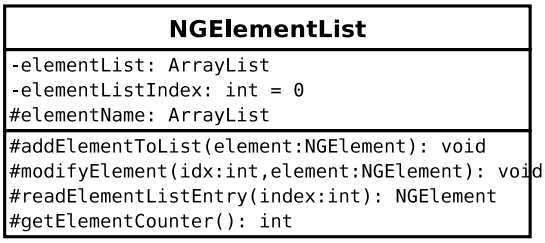


Abbildung 7.9: Detailansicht Klassendiagramm NGElementList

Die Anzahl der Attribute und Methoden ist logischerweise recht groß, da für jeden erdenklichen Parameter und Typ eines Gleisnetzelements hier eine Erfassungsmöglichkeit vorhanden sein muss und die Objekte dieser Klasse für jedes Element in einem Gleisnetz genutzt wird.
Für jede Eigenschaft existiert also ein Attribut, eine Methode zum Setzen dieses Attributs (Set-Methode) und eine Methode, um die Daten dieses Attributes wieder auslesen zu können (Get-Methode).
Außerdem sind einige konstante Parameter definiert, dessen Werte in einzelnen Attributen abgespeichert werden können, wenn diese für den entsprechenden Element-Zustand und -Typ zutreffend sind, wie zum Beispiel der Typ einer Weiche.

7.2.3.2.8 NGElementList.java Die zuvor genannten NGElement-Objekte werden zur einfacheren Abarbeitung in *ArrayLists* gespeichert. Die Klasse *NGElementList.java* stellt solche ArrayLists und entsprechende Methoden zur Bearbeitung derselben bereit. Die NGElementList-Objekte dienen letzlich auch als eine Art Sortier-Container, aus denen später beim Erzeugen die Gleisnetzelemente nach globalen Typen sortiert ausgelesen werden können.
Das detaillierte Klassendiagramm (Abb. 7.9) zeigt die in dieser Klasse verwendeten Methoden und Attribute.

Im Gegensatz zu *NGElement.java* ist die Anzahl der Attribute und Methoden hier überschaubar, da hier letzlich nur andere Objekte gesammelt werden.

7.2.3.2.9 NGLines.java Neben den Elementen eines Gleisnetzes, wie zum Beispiel Weichen oder Signalen gibt es auch noch die Verbindungsstücke dieser Elemente - die

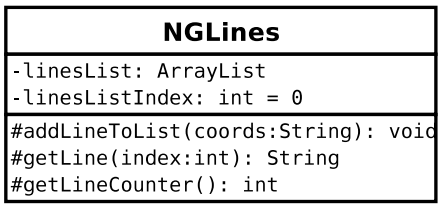


Abbildung 7.10: Detailansicht Klassendiagramm NGLines

einfachen Gleisstücke.
Entgegen anfänglicher Planungen, diese als eigene Elemente (in der CAD-Objektbibliothek) zu realisieren, werden in der Endfassung nun simple Linien variabler Länge als Gleisstück benutzt. Da diese Linien für die Zusammensetzung eines Gleisnetzes eine relevante Rolle spielen, indem sie die Nachbarschaftsbeziehungen einzelner Elemente spezifizieren, werden alle Verbindungslinien ebenfalls in einem ArrayList-Objekt gespeichert.
In diesem Zusammenhang ist die Nebenklasse *NGLines.java* entstanden, welche neben der ArrayList auch Methoden zur einfacheren Verarbeitung liefert.
Das detaillierte Klassendiagramm (Abb. 7.10) zeigt die in dieser Klasse verwendeten Methoden und Attribute.

Auch hier ist aus einem ähnlichen Grund wie bei *NGElementList.java* der Umfang der Attribute und Methoden schnell überschaubar.

7.2.3.2.10 TNDCreator.java Die dritte Hauptklasse (*TNDCreator.java*) in dieser Idee ist für die eigentliche Generierung der TND-Beschreibung des Gleisnetzes zuständig. Diese Klasse ruft letztlich die bereits beschriebene Klasse *NGDataCollector.java* auf und liest aus den damit erzeugten Sortier-Containern - den *NGElementList*-Objekten - die einzelnen Objekte vom Typ *NGElement* aus, um mit deren Daten die einzelnen Blöcke der zu erzeugenden TND-Datei zusammenzusetzen. Die Daten der in den Objekten gespeicherten Elemente müssen dabei noch etwas aufbereitet werden, damit sie dem in der TND geforderten Format entsprechen.

Da in der TND-Datei die Koordinaten als Zentimeter-Angabe definiert sind, in der DXF-Datei aber ein anderes Maß verwendet wird, muss die Angabe aller Koordinaten entsprechend umgerechnet werden. Als Faktor wurde hier der Wert 100 gewählt, da so die Auflösung innerhalb der DXF-Datei in einem sinnvollen Maßstab gehalten wird und die Werte nicht großartig verändert werden müssen. Ein Weichenelement, welches von der Einfahrtsseite zur gegenüberliegenden Ausfahrtsseite somit in der DXF-Datei eine relative Länge von 10 aufweist würde somit in der Realität eine Länge von 10 Metern - also 1000 Zentimetern haben. Dies erscheint als realistischer Wert. Letzlich ist dies aber

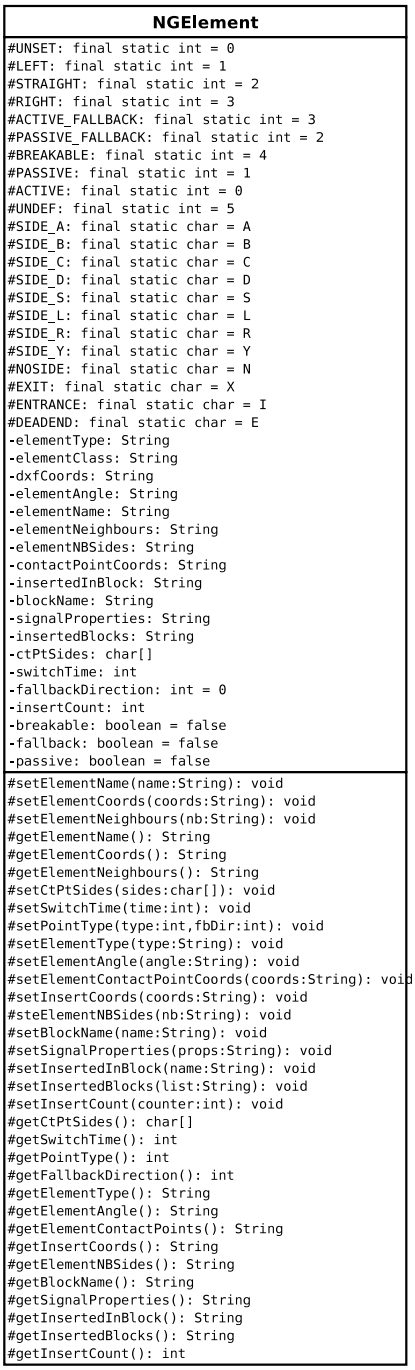


Abbildung 7.11: Detailansicht Klassendiagramm NGElement

individuell im Quelltext anpassbar und sollte die eigentliche Funktion in keiner Weise beeinträchtigen.

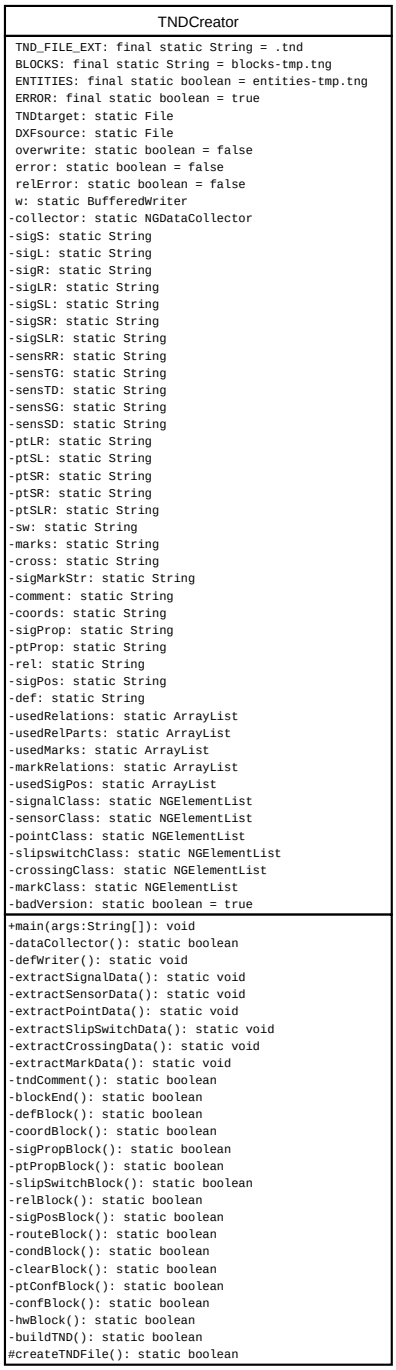


Abbildung 7.12: Detailansicht Klassendiagramm TNDCreator

In der TND ist es auch wichtig, die verschiedenen Typen von Signalen, Sensoren, etc. voneinander zu unterscheiden, so dass diese Arbeit auch in dieser Klasse während des

Erzeugens der TND-Beschreibung erledigt wird, da bisher nur eine Grobsortierung (z.B. alle Signale) in globale Kategorien vorgenommen wurde. In den Objekten sind bereits alle notwendigen Daten enthalten, aber es wurde eben bisher nur eine Vorsortierung vorgenommen.
Auch werden Signale im Gegensatz zu anderen Elementen in besonderer Weise in der TND-Darstellung repräsentiert, so dass auch hier noch etwas an Aufbereitungsbedarf besteht, bevor im nächsten Schritt die Blöcke zu einer TND-Datei zusammengesetzt werden können. Auch dies wird hier gemacht.
Intern wird die TND als eine Aneinanderreihung von diversen Zeichenketten repräsentiert, welche im finalen Schritt in die zu erzeugende Datei geschrieben werden.
Da aus dem Netzgraphen nicht alle für ein steuerbares Gleisnetz benötigten Informationen gewonnen werden können (genau betrifft dies die Routeninformationen und die Hardwareklassendefinitionen), werden jene Blöcke leer gelassen, so dass die TND auch ohne diese Informationen einer formalen Gültigkeit entspricht. Sinnvoll steuerbar ist das so dargestellt Gleisnetz allerdings noch nicht, so dass die dazu fehlenden Informationen von einer weiteren Teil-Komponente geliefert werden müssen (siehe dazu *TND-Builder*, 7.3 auf Seite 123).

Die Detailansicht des Klassendiagramms für diese Klasse (Abb. 7.12 auf der vorherigen Seite) zeigt die zur Verwendung kommenden Methoden und Attribute.
Die Anzahl der Attribute ist im Vergleich zu der der Methoden recht groß, da wie beschrieben die TND intern als eine Aneinanderreihung von Zeichenketten repräsentiert wird. Die diversen String-Attribute stellen eben jene Zeichenketten her, welche nach und nach beim Auslesen der Objekte aus den Element-Containern (ArrayLists) gefüllt werden.
Dies ist erforderlich, da ein Objekt (Gleisnetzelement) Informationen für mehrere Teile der TND enthält (zum Beispiel Koordinaten und bestimmte Properties (Eigenschaften), etc.).
Die Methoden sind zum Teil für das Auslesen der Sortier-Container da (z.B. *extractSignalData()* für die evtl. vorkommenden Signale). In diesen Methoden wird dann auch die oben beschriebene letzte Aufbereitung der Daten vollzogen.
Die Methode *extractSlipSwitchData()* ist nur in der Test-Version verfügbar, da sie nicht ausreichend getestet werden konnte (siehe dazu Abschnitt 7.2.3.2.13 auf Seite 103) .
Ein andere Teil der Methoden ist für das Zusammensetzen der TND-Datei als solches zuständig. Für jeden einzelnen Block existiert eine Methode (z.B. *coordBlock()* für den Koordinaten-Block).
In *buildTND()* wird die TND dann endgültig zusammengesetzt.
Um aber überhaupt die Daten des Gleisnetzes zu erhalten, muss zuerst das „Sammeln“ der Daten initiert werden, indem die Klasse *NGDataCollector.java* instanziiert wird. Dies geschieht mit Hilfe der Methode *dataCollector()*.
Die Methoden sind zu einem großen Teil vom Typ **boolean**, womit dort auftretende

Fehler leicht gemeldet und ggf. die weitere Ausführung abgebrochen werden kann.

7.2.3.2.11 TracsDXF2TND.java Um nicht mehrere verschiedene Aufrufe tätigen zu müssen, um eine CAD-Zeichnung im DXF-Format in die TND-Gleisnetzbeschreibung zu überführen, wurde eine übergeordnete Klasse *TracsDXF2TND.java* geschaffen, welche letzlich nichts anderes macht, als die drei genannten Hauptklassen der Reihe nach aufzurufen.
Dieser Klasse wird genau wie der Klasse *DXFSeparator.java* eine DXF-Datei übergeben. Zusätzlich wird aber auch das Ziel - eine TND-Datei angegeben.
Eine weitere Funktion ist das Aufrufen einer Auflistung sämtlicher Gleisnetzelemente (Anhang E auf Seite 475 durch Angabe eines optionalen Parameters.
Die genauen Aufruf-Parameter werden - wie auch die einzelnen Arbeitsschritte bei der Gleisnetzerfassung und TND-Erzeugung aus diesen Daten - im Handbuch des Konverters erläutert (siehe Abschnitt 7.2.3.4 auf Seite 104).
Die Detailansicht des Klassendiagramms für diese Klasse (Abb. 7.13) zeigt die hierbei verwendeten Methoden und Attribute.

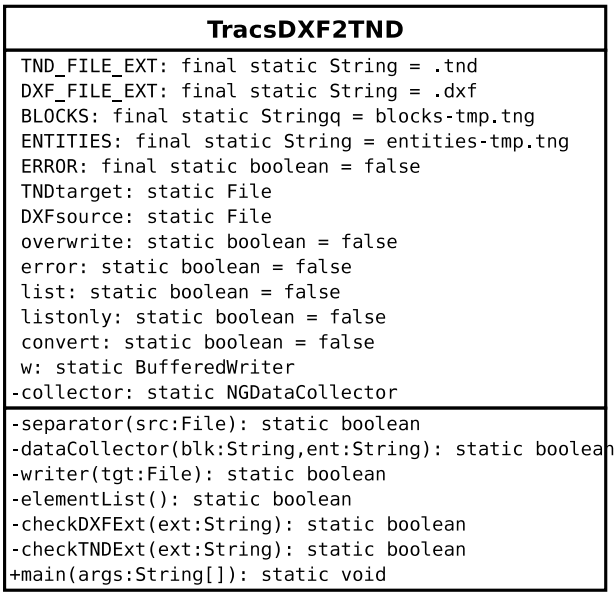


Abbildung 7.13: Detailansicht Klassendiagramm TracsDXF2TND

Es werden unter anderem für die Instanziierung der einzelnen Komponenten entsprechende Methoden zur Verfügung gestellt. All diese Methoden sind nicht-öffentlich, da sie nur innerhalb dieses Programms, bzw. dieser Klasse Verwendung finden.

7.2.3.2.12 NGSystem.java Um sicherzustellen, dass der Konverter nur in einer Software-Umgebung läuft, die der getesteten ähnelt, wurde eine weitere Hilfsklasse *NGSystems.java* erstellt.
Diese Klasse enthält eine Prüfmethode, welche sowohl verwendetes Betriebssystem, als auch die Java™-Umgebung überprüft und bei einer nicht getesteten Version das Programm mit einer entsprechenden Fehlermeldung abbricht.
Der Grund für die Einführung dieser Klasse waren Unterschiede oder besser gesagt Fehler auf einer Plattform, während auf einem anderem System und ansonsten gleichen Daten und Klassen keine Fehler auftraten. Da diese Fehler nicht offensichtlich waren, wurde aus zeitlichen Gründen eine Einschränkung auf das funktionierende System vorgenommen.

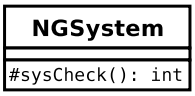


Abbildung 7.14: Detailansicht Klassendiagramm NGSystem

7.2.3.2.13 Zusätzliche Test-Version Da kurz vor Projektende doch noch ein paar zusätzliche, bisher aus Zeitgründen nicht implementierte Funktionalitäten hinzugefügt wurden, diese aber bei weitem nicht ausreichend getestet werden konnten, existieren zwei Versionen des Konverters.
Die Test-Version verfügt im Gegensatz zur stabilen Version über eine eingeschränkte Erkennung und Konvertierung von Slipswitches. Die Einschränkung hierbei ist allerdings, dass diese Elemente in der CAD-Zeichnung nicht gedreht werden sollten, da sonst die Koordinaten der internen Einfachweichen nicht korrekt sind. Hierzu war am Ende leider definitiv keine Zeit mehr übrig.
Eine zweite Verbesserung gegenüber der stabilen Version sind Mehrfach-Markierungen (mehrere Markierungen in Folge). Mit dieser Verbesserung können nun auch Kurven modelliert werden.
Ansonsten sind beide Versionen in ihrer Funktionalität identisch.

7.2.3.3 TND-CAD-Rück-Konverter

Ein drittes ursprünglich geplantes Arbeitspaket des Netzgraphen war ein TND-CAD-Rück-Konverter. Dieser sollte aus der EBNF-Beschreibung der TND wieder eine Visualierung im CAD-System herstellen. Da sich aber mit der Zeit herausgestellt hat, dass der Simulator ebenfalls eine Visualisierung des in der TND beschriebenen Gleisnetzes realisiert, wurde auch in Anbetracht des im Laufe der Zeit immer begrenzter zur Verfügung

stehenden Personals und des daraus resultierenden Zeitmangels beschlossen, dieses Arbeitspaket ersatzlos zu streichen.
Es wäre allerdings auch extrem aufwändig gewesen, dieses Paket zu realisieren, da, wie bereits mehrfach erwähnt, das DXF-Format, in welchem die CAD-Software eine Zeichnung abspeichert, in extrem vielen Versionen verfügbar ist und jede Version diverse Zusatzinformationen enthält, welche nicht wirklich von einem Rückkonverter generiert werden könnten. Ob eine DXF-Zeichnung mit den Minimal-Informationen wirklich einwandfrei im CAD-Programm darstellbar gewesen wäre, ist unklar.
Auf jeden Fall wären deutlich mehr Personen in der Netzgraphgruppe dafür notwendig gewesen, als letztlich zur Verfügung standen.

7.2.3.4 Handbuch zu den Konverter-Tools

Das abschließende Arbeitspaket im Teilprojekt Netzgraph war das Anfertigen eines Handbuchs für den Netzgraph-Konverter inklusive einer Anleitung für die Erzeugung einer Basis-TND-Datei. Dieses Handbuch umfasst sowohl die eigentlichen Aufrufparameter des Konverters, als auch Informationen zur Installation als solches.
Der wesentliche Bestandteil aber ist die genaue Erläuterung, wie ein Zeichner eines Gleisnetzes dieses mit Hilfe der CAD-Software und der von TRACS gelieferten Objekt-Bibliothek anfertigt, was zu beachten ist und auch welche Einschränkungen es in der vorliegenden Version hinsichtlich des Weltmodells und der Möglichkeiten der TND gibt. Das Handbuch ist im Abschnitt 7.2.4 zu finden.
Da dieses Handbuch eigentlich als eigenständiges Dokument gedacht war, welches der Software beiliegt, können sich Wiederholungen in der Beschreibung im Hinblick auf einzelne Teile dieses Berichtes ergeben.

Henrik Röhrup

7.2.4 Benutzerhandbuch DXF/CAD-nach-TND-Konverter

Dieses Dokument soll eine Einführung in den DXF/CAD-nach-TND-Konverter liefern. Es werden Informationen zu der zum Betrieb erforderlichen Systemumgebung geliefert, wie auch weitere Hinweise zum Erstellen einer TND-Datei, wie sie von anderen zum TRACS-Projekt gehörenden Komponenten benutzt wird.

7.2.4.1 Überblick

Der DXF/CAD-nach-TND-Konverter ist eine eigenständige Software, welche aus einer mit Hilfe der mitgelieferten Objekt-Bibliothek erstellten CAD-Datei in DXF-Form ein Gleisnetz in TND-Form generiert.
Diese TND enthält alle notwendigen Informationen, um das Gleisnetz zum Beispiel mit

dem TRACS-Simulator, der TND-Dateien einlesen kann, darstellen zu können.
Damit ein solches Gleisnetz auch von einer Steuerungssoftware gesteuert werden kann, müssen jedoch noch weitere Informationen hinzugefügt werden. Diese Informationen können aber nicht aus dem Netzgraphen eines Straßenbahnnetzes abgeleitet werden und müssen durch zusätzliche Tools ergänzt werden, welche diese Basis-TND dazu einlesen können.
In den folgenden Abschnitten werden zum einen die Systemvorausetzungen und zum anderen die Vorgehensweisen zur Benutzung dieses Konverters beschrieben. Außerdem wird erklärt, wie ein Gleisnetz mittels einer CAD-Software (QCad) und der mitgelieferten TRACS-Objekt-Bibliothek erstellt werden kann.
In diesem Software-Paket befinden sich zwei Versionen des Konverters, da ein paar Funktionen nicht ausreichend getestet werden konnten. Diese sind also nur in der Test-Version verfügbar. Genaueres ist im Abschnitt zu den Einschränkungen bei der TND-Erzeugung zu finden.

7.2.4.2 Installation und Systemvoraussetzungen

Generell ist anzumerken, dass die Software auf einem Linux-System mit aktuellem Kernel der 2.6er Reihe entworfen wurde und auf jedem vergleichbaren System lauffähig sein sollte.

7.2.4.2.1 Systemvoraussetzungen Sowohl die benötigte **Java™**-Entwicklungsumgebung, bzw. Laufzeitumgebung, als auch die CAD-Software **QCad** sind sowohl für **Linux**, als auch für **Microsoft®Windows™** verfügbar. Getestet wurde die Funktion aber nur auf einem Linuxsystem und einer i386-kompatiblen PC-Plattform und daher wird in der vorliegenden Version die Ausführung auf anderen Plattformen auch unterbunden.
Bei der Entwicklung wurde ein Linux-System mit Kernel der Version 2.6 - genau ein Debian GNU/Linux 3.1r0 mit Kernel 2.6.8 (auch als Debian Sarge bekannt) - genutzt. Der Rechner selbst war ein PC mit einer AMD Athlon-CPU und 500 MHz Taktrate, sowie einem Hauptspeicher von 256 MB. Erfolgreich getestet wurde das System aber auch auf einem Intel Pentium-Prozessor mit 166 MHz und 64 MB Hauptspeicher.
Als Entwiklungsumgebung wurde das Java2™-Software-Developmentkit (J2SDK) in der Version 1.4.2 verwendet. Eine J2SDK oder J2RE (Java2™-Runtime-Environment) der Version 1.4.x der Firma **Sun Microsystems Inc.** sollte auf jeden Fall verwendet werden. Java™-Umgebungen anderer Hersteller wurden nicht getestet und daher in dieser Version nicht unterstützt. Ob die Software auch unter einer anderen Version prinzipiell lauffähig ist, wird nicht garantiert.
QCad wurde in Version 2.0.4.0 (www.ribbonsoft.com) verwendet, welche zusammen mit dem verwendeten Linux-System mitgeliefert wurde. Eine Version kleiner 2.x wird nicht unterstützt, da wesentliche Funktionalitäten fehlen. Ob auch andere DXF-kompatible

CAD-Software, wie zum Beispiel AutoCAD nutzbar ist, kann nicht klar gesagt werden, da TRACS keine Alternative zur Verfügung stand. Wenn der Funktionsumfang aber mindestens dem von QCad 2.x entspricht und mit einer DXF-Element-Bibliothek in dem Stil, wie sie von QCad verwendet wird, gearbeitet werden kann, so sollte dies möglich sein.

7.2.4.2.2 Installation Es wird davon ausgegangen, dass auf dem System sowohl QCad, als auch die Java™-Laufzeitumgebung (J2RE) bzw. die Entwicklungsumgebung (J2SDK) installiert sind.
Die TRACS-Objekt-Bibliothek für die CAD-Software sollte idealerweise in jenem Verzeichnis abgelegt werden, welches für die sogenannten Teile-Bibliotheken vorgesehen ist. Auf dem verwendeten System, war dies der Pfad **/usr/share/qcad/libraries**.
Theoretisch sollte es aber auch möglich sein, ein beliebiges Verzeichnis zu wählen und in den Einstellungen der CAD-Software den Pfad entsprechend zu verändern.
Was die Konverter-Tools betrifft, so reicht es, diese in einem beliebigen Verzeichnis abzulegen, auf das während der gesamten Laufzeit Schreibzugriff für den ausführenden Benutzer besteht.
Sollten die einzelnen Java-Klassen erst noch übersetzt werden müssen, so reicht es, in den Pfad zu wechseln, welches das Verzeichnis mit dem Namen **ng** enthält, in welchem sämtliche Java-Klassen des Konverters zu finden sind und mittels **javac ng/*.java** diese zu übersetzen. Als Ergebnis erhält man die mittels des Befehls **java** ausführbaren Programme.

7.2.4.3 Benutzung der CAD-Software QCad und der Objekt-Bibliothek

Für die erfolgreiche Erstellung von Gleisnetzdarstellungen in einer CAD-Software und anschließender Konvertierung wird das Programm QCad in einer Version 2.x benötigt. Dieses Programm ist für verschiedene Betriebssysteme vorhanden. Da der TracsDXF2-TND-Konverter in erster Linie für ein Linux-System entwickelt wurde, wird dazu geraten, auch die entsprechende Linux-Version von QCad zu verwenden.
QCad sollte in jeder aktuellen Linux-Distribution enthalten sein. Empfohlen wird von uns **Debian GNU/Linux 3.1**.

7.2.4.3.1 Einbinden der Objekt-Bibliothek Um entsprechende Gleisnetze zeichnen zu können, muss zu allererst die von TRACS gelieferte Objekt-Bibliothek in QCad eingebunden werden. Dazu muss der Pfad zu dieser Bibliothek in QCad gespeichert werden.
Dies ist über das Programm-Menü „Bearbeiten“ und dem Menüpunkt „Applikations-Einstellungen“ und dann auf der Seite „Pfade“ zu erldigen. Hier befindet sich der Eintrag „Teile Bibliotheken“, wo der entsprechende absolute Pfad zu setzen ist. (siehe auch Ab-

bildung 7.15)

Abbildung 7.15: Eingabe des Bibliothekspfades

Sollte es vorkommen, dass diese Einstellung nicht gespeichert wird, so ist zu empfehlen, den dort vorgegebenen Pfad zu verwenden und die Objekt-Bibliothek unterhalb dieses Pfades abzuspeichern. Auf diese Weise sollte auf jeden Fall die Verfügbarkeit sichergestellt sein.
Desweiteren sollte über den Menüpunkt „Ansichten“ des Menüs „Ansicht“ der „Bibliotheks-Browser“ (Abbildung 7.18 auf Seite 109) aktiviert werden.

7.2.4.3.2 Nutzung der TRACS-Objekt-Bibliothek Das Haupt-Programm-Fenster von QCad (Abbildung 7.16 auf der nächsten Seite) beinhaltet ein größeres Unterfenster, in dem die Zeichnung angefertigt wird, sowie in der Standard-Einstellung auf der linken Seite eine Werkzeugleiste (Abbildung 7.17 auf der nächsten Seite). Auf der rechten Seite ist unter anderem der Bibliotheks-Browser zu sehen.
Im Bibliotheks-Browser von QCad (siehe auch Abbildung 7.18 auf Seite 109) sind nun

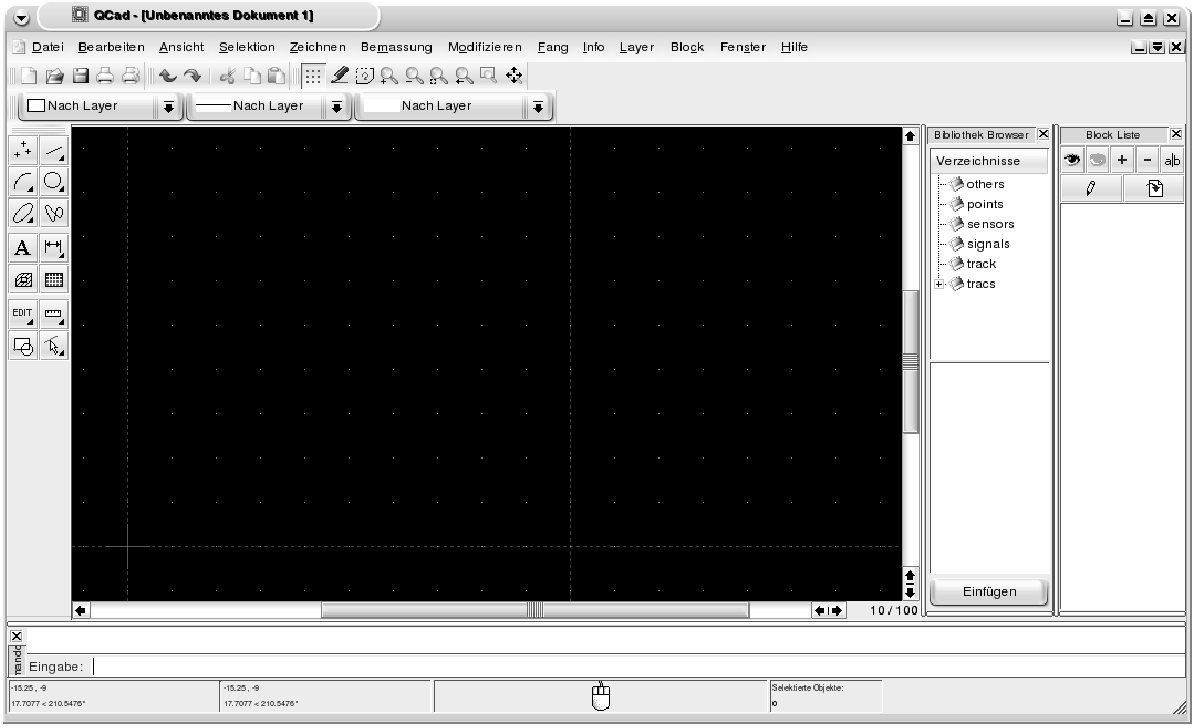


Abbildung 7.16: QCad-Hauptfenster

die einzelnen Gleisnetzelement-Zeichnungen auswählbar und können nach Auswahl und Klick auf „Einfügen“ in der Zeichnung an beliebiger Stelle per Mausklick plaziert werden.

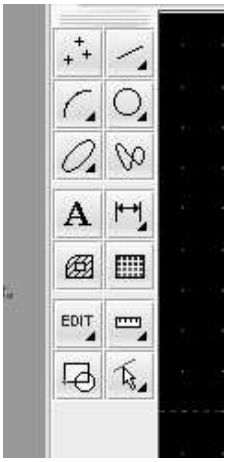


Abbildung 7.17: QCad-Werkzeugleiste

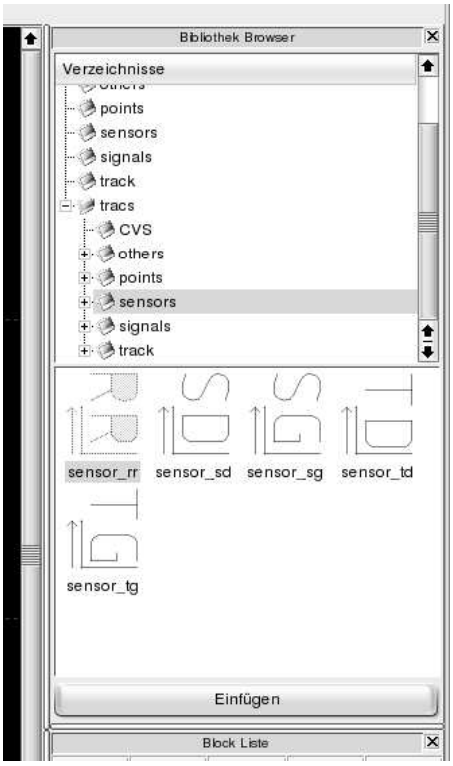


Abbildung 7.18: QCad-Bibliotheks-Browser

Die einzelnen Elemente sind in der Objekt-Bibliothek in verschiedene Kategorien unterteilt, wobei deren Namen in den meisten Fällen selbsterklärend sein sollten. Die Kategorie „track“ umfasst alle Schienen-Elemente, die keiner anderen Kategorie zuzuordnen sind - also z.B. Kreuzungen, während „points“ alle Weichen-Elemente enthält. Die Kategorien „signals“ und „sensors“ enthalten ausschließlich die im Namen genannten Elemente. In der Kategorie „others“ sind Markierungselemente zu finden, sowie Symbole für Eintritts- und Ausgangspunkte eines Gleisnetzes.
Einfache Gleise werden durch simple Linien dargestellt. Hierfür gibt es keine eigenen Teilzeichnungen innerhalb der Bibliothek.

7.2.4.3.3 Hinweise zum Zeichnen mit QCad

Rasterfang:
Es sollte auf jeden Fall darauf geachtet werden, dass die Funktion „Raster fangen“ aktiviert ist. Dies ist z.B. über das Menü „Fang“ einstellbar.
Diese Funktionalität ist erforderlich, weil es exakte Berührungspunkte zwischen zwei Elementen oder einem Element und einer Linie (einfaches Gleisstück) gibt. Diese müssen genau übereinanderliegen, da sonst keine Nachbarschaftsbeziehungen erkannt werden

können, was aber für die spätere Konvertierung in das TND-Format zwingend notwendig ist.

Dateiformat:
Als Dateiformat beim Speichern ist **DXF** zu wählen. Hierbei ist es irrelevant, welche Version dieses Formats gewählt wird.

7.2.4.3.4 Zeichnen eines Gleisnetzes Beim Zeichnen eines Gleisnetzes ist darauf zu achten, dass sich alle Elemente mit ihren Nachbarn in jeweils einem Punkt berühren müssen - sog. Berührungspunkte. Diese Berührungen dürfen nicht an beliebiger Stelle eines Elements stattfinden, sondern müssen exakt an den Enden erfolgen.
Es darf außerdem kein Ende eines Elements frei bleiben, d.h., dass es keine offenen Enden geben darf. Ein- und Ausfahrten eines Gleisnetzes müssen dann ggf. mit einer entsprechenden Markierung versehen werden, damit das Gleisnetz korrekt ist.
Linienzüge (also mehrere Linien in Folge ohne weitere Elemente dazwischen) als Verbindung zweier Elemente sind auch nicht gestattet, stattdessen muss immer eine einzelne Linie zwei Elemente oder ein Element und eine Markierung verbinden.
Ein Gleisnetz muss immer mindestens einen Eingang und mindestens einen Ausgang haben, welche durch entsprechende Markierungssymbole definiert sind. Diese Symbole bilden den Abschluss eines Gleises und es darf folglich immer nur ein Element, welcher Art auch immer mit diesen Symbolen verbunden werden.
Desweiteren dürfen sich auch nicht mehrere Linien oder Elemente in einem und demselben Berührungspunkt eines anderen Elements treffen. Pro Berührungspunkt darf es nur zwei Elemente geben. Ausnahme bilden hier Markierungen, da diese auf dem Berührungspunkt zweier Linien liegen und somit quasi drei Teile sich in diesem Punkt treffen.
Markierungen sind außerdem nur zwischen zwei Linien erlaubt - nicht aber direkt an einem anderen Element.
Sensoren und Signale sind gerichtete Elemente, so dass ggf. auf die Fahrtrichtung geachtet werden muss. Bei diesen Elementen ist außerdem darauf zu achten, dass es sich wegen der Fahrtrichtungserkennung um Zwei-Punkt-Elemente handelt, deren zwei Koordinaten sehr eng beieinander liegen, sodass es zu Zeichenfehlern kommen könnte, wenn man nicht genau aufpasst. Eine Vergrößerung des betreffenden Ausschnitts (Heranzoomen) ist daher ratsam.

Benennung von Elementen:
Die einzelnen Elemente eines Gleisnetzes müssen zum Zwecke der Konvertierung und korrekten Erstellung einer TND-Datei eindeutige Namen haben. Außerdem müssen ggf. bei Weichen und Signalen einzelne Properties (Eigenschaften, wie z.B. besondere Signal-Anzeigen) gesetzt werden. Dies wird beides im jeweils selben Arbeitsschritt

durchgeführt.
Um ein Element eineutig entsprechend der Syntax der TND zu benennen, muss irgendwo in relativer Nähe ein Textfeld plaziert weren, welches den Namen enthält. Müssen auch noch zusätzliche Properties eingegeben werden, so geschieht dies im selben Textfeld. Die Properties werden von sogenannten „Unterstrichen“ vom Namen des Elements abgetrennt, wobei die *Switchtime* immer die erste Eigenschaft ist. Die Syntax lautet wie folgt:

```
-Signale: Elementname_Switchtime_Signal-Properties
-Weichen: Elementname_Switchtime
-sonst. Elemente: Elementname

Beispiel eines L-Signales mit der Schaltzeit von 500 ms und
Anzeigen für Wait-Left, Route-Request-Left:

s359_500_RRL_WL

Beispiel einer LR-Weiche mit der Schaltzeit von 700 ms:

w467_700
```

Die Switchtime (bei Signalen und Weichen) ist ein Wert in Millisekunden. Die Signal-Properties (selbstredend nur bei Signalen) sind aus einer für den entsprechenden Signal-Typ möglichen Menge zu wählen (RRL, RRR, RRS, WL, WR, WS - RRx sind Route-Request-Properties (Routen-Anforderungs-Anzeigen) während Wx für Wait-Properties (Anzeigen für Warte-Aufforderungen) steht).
Damit dieses Textfeld aber exakt einem Element zugeordnet werden kann, muss es mit dem entsprechenden Element-Block zu einem neuen Block kombiniert werden. Dies kann mittels des Menüpunktes „Block erstellen“ aus dem Menü „Block“ gemacht werden:
Für den Namen dürfen nicht beliebige Zeichen verwendet werden. Je nach Elementtyp wird ein bestimmter Anfangsbuchstabe (und zwar klein geschrieben!) gefordert. Anschließend muss mindestens ein alpha-numerisches Zeichen folgen. Die Anzahl ist beliebig, wobei aber eben nur Buchstaben (a-z) und Ziffern (0-9) verwendet werden dürfen - sonst nichts!
Für Weichen wird als Anfangsbuchstabe ein „w“ verwendet. Für Signale muss ein „s“, für Sensoren ein „g“, für Kreuzungen ein „k“, für Slipswitches„q“ (wobei Slipswitches nur in der Test-Version unterstützt werden) und für Markierungen ein „x“ gewählt werden.
Aufgrund eines internen Designs bei Signal-Darstellungen sollte man es vermeiden, Markierungen mit der Anfangs-Zeichenkette „xs“ zu verwenden.
Dasselbe gilt für Bezeichner mit „kq“ oder „wq“ am Anfang, da Slipswitches in Kreuzungen und Weichen aufgelöst werden (sofern sie unterstützt werden).

Signale werden in der später zu generierenden TND-Datei auf Markierungen aufgesetzt, so dass diese besonderen Markierungen letzlich die ID (den Namen) des Signals mit einem vorangestellten „x“ als eigene Bezeichnung tragen.
Sowohl Text, als auch zugehöriges Element müssen markiert werden. Danach wird auf das untere schwarze Pfeilsymbol in der Werkzeugleiste (siehe auch Abbildung 7.19) geklickt und anschließend ein beliebiger Bezugspunkt gewählt. Abschließend wird ein beliebieger Name für diesen neuen Block gewählt. Der Name und der Bezugspunkt haben für die spätere Konvertierung keine besondere Bedeutung, sondern sind nur für die interne Darstellung der DXF-Datei von Relevanz. Der Name muss aber eindeutig sein.

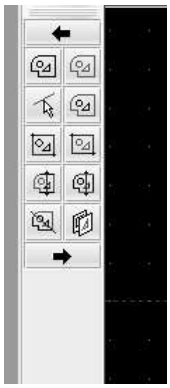


Abbildung 7.19: QCad-Werkzeugleiste Blockerstellung

Kurven:
Es ist möglich, Kurven zu zeichnen, indem mehrere kurze gerade Linien aneinander gesetzt und auf diesen Knotenpunkten einfache Markierungen platziert werden, welche natürlich auch benannt werden müssen.
Diese Funktionalität ist aber nur in der Test-Version des Konverters fehlerfrei nutzbar.

Layer-Darstellung:
Es ist möglich, zur besseren Übersicht einzelne Elementtypen auszublenden. Dazu wurden die einzelnen Elementtypen auf verschiedenen Ebenen (Layern) erstellt. Beim Einfügen eines neuen Elements, sollte man sich immer auf der Hauptebene befinden, wenn man denn einen sinnvollen Ensatz der Layer-Technik sicherstellen will. Für die Konvertierung ist dies aber nicht erforderlich, sondern lediglich als Hilfestellung bei komplexeren Gleisnetzen gedacht, um den Überblick zu behalten.
Auf Details wird hier verzichtet, da dies nicht von wesentlicher Bedeutung ist und wir davon ausgehen, dass sich der Benutzer mit der CAD-Software als solches vertraut macht. Informationen zu QCad sind im Internet unter der Adresse **www.ribbonsoft.com** zu finden. Dort ist zum einen das Programm selbst downloadbar, als auch eine Dokumen-

tation zu finden.

7.2.4.4 Bedienung der Konverter-Tools

Der DXF/CAD-nach-TND-Konverter kann auf verschiedene Weise aufgerufen werden. So ist es zum Beispiel möglich für Debuggung-Zwecke die jeweiligen Schritte einzeln manuell zu starten. Oder aber es wird eine DXF-Datei in einem Durchgang in eine Basis-TND-Datei überführt (was die empfohlene Methode ist). Es ist außerdem möglich sich eine komplette Liste aller Elemente in Textform ausgeben zu lassen.
Im Folgenden sollen nun die einzelnen Aufrufe beschrieben werden. Es wird dabei angenommen, dass man sich in der Verzeichnisebene befindet, in der das Verzeichnis **ng** zu finden ist, welches sämtliche Programmdateien des Konverters enthält.
Die einzelnen Teilprogramme sind: **DXFSeparator.java**, **NDDatacollector.java** und **TNDCreator.java**. **TracsDXF2TND.java** ist jenes Program, welches alle Schritte in einem vereint.

7.2.4.4.1 DXFSeparator Der erste Schritt in der Konvertierung ist das Separieren der für den Konvertiervorgang relevanten von den irrelvanten Informationen einer DXF-Datei. Diese Trennung wird durch das Programm **DXFSeparator.java** durchgeführt.
Der Aufruf ist ganz einfach: Es wird dem Programm einfach die zu konvertierende DXF-Datei übergeben und man erhält (im aktuellen Verzeichnis) zwei Zwischendateien mit den Namen **blocks-tmp.tng** und **entities-tmp.tng**.
Diese beiden Dateien enthalten die beiden Sektionen einer DXF-Datei, welche für uns von Relevanz sind. Außerdem sind in dem Trennvorgang schon einige zusätzliche nicht benötigte Informationen aus den beiden Sektionen entfernt worden.
Der genaue Aufruf lautet:

```
java ng\DXFSeparator <DXF-File>
```

7.2.4.4.2 NGDataCollector Der zweite Schritt ist das Extrahieren der Gleisnetzdaten aus diesen Zwischendateien. Dies geschieht durch das Programm **NGDataCollector.java**, welches die DXF-Informationen soweit zerlegt, so dass daraus die einzelnen Elemente eines Gleisnetzes bestimmt werden können.
Diese Elemente werden intern in sogenannten Objekten gespeichert und in Listen nach Kategorien sortiert.
Diesem Programm werden die im vorangegangenen Schritt erzeugten Zwischendateien übergeben und man erhält eine Liste aller im Gleisnetz enthaltenen Elemente.
Der genaue Aufruf lautet:

```
java ng\NGDataCollector <blocks-tmp.tng> <entities-tmp.tng>
```

7.2.4.4.3 TNDCreator Der dritte und letzte Schritt ist das Erzeugen der TND-Datei. Dies geschieht mit Hilfe des Programms **TNDCreator.java**. Diesem Programm wird als Parameter einfach der Name der Zieldatei - nämlich der zu erzeugenden TND-Datei übergeben. Es wird dabei angenommen, dass sich jene Zwischendateien mit exakt den Namen, wie sie erzeugt wurden im aktuellen Verzeichnis befinden.
Dieses Programm ruft die zuvor erwähnte Klasse auf, um an die eigentlichen Element-Daten zu gelangen. Dabei werden die Elemente ausgelesen, umgewandelt und der Output in die angegebene TND-Datei geschrieben.
Der genaue Aufruf lautet:

```
java ng\TNDCreator <TND-File>
```

7.2.4.4.4 TracsDXF2TND Die zuvor beschriebenen Schritte sind lediglich für ein manuelles Debugging sinnvoll. Im Normalfall sollte der Konvertiervorgang mit einem einzigen Programmaufruf durchgeführt werden. Dazu gibt es das Programm **TracsDXF2TND.java**, welches alle zuvor beschriebenen Schritte vereint.
Es werden dem Programm sowohl die DXF-Quell-Datei, als auch die TND-Ziel-Datei als Parameter übergeben. Optional kann noch ein dritter Parameter „–list“ angegeben werden, welcher das Anzeigen der Elementliste ermöglicht.
Der genaue Aufruf lautet:

```
java ng\TracsDXFTND {--list} <DXF-File> <TND-File>
```

Soll nur die Liste ausgegeben werden, aber keine Erzeugung der TND durchgeführt werden, kann einfach der Parameter für die TND-Datei weggelassen werden.
Externe Programme, welche zur Bearbeitung der TND eingesetzt werden, um zum Beispiel die Routeninformationen hinzuzufügen, sollten eben diesen Befehl einbinden, um die DXF-Datei mehr oder weniger direkt einzulesen.
Durch Abfrage des Exit-Codes kann eine erfolgreiche Programm-Ausführung festgestellt werden. Ein Exit-Code ungleich „0“ bedeutet einen Fehler, während „0“ selbst eine erfolgreiche Ausführung bedeutet.
Ein Beispiel-Code für den Aufruf aus einem anderen Java™-Programm ist hier zu sehen:

```
    try {
        Runtime rt = Runtime.getRuntime();
        Process proc = rt.exec("java ng/TNDTracsDXF2TND test.dxf
                       test.tnd");
        InputStream stderr = proc.getErrorStream();
        InputStreamReader isr = new InputStreamReader(stderr);
        BufferedReader br = new BufferedReader(isr);
//        String line = null;
//        while ((line = br.readLine()) != null) {
```

```
//            System.out.println(line);
//        }
        int exitVal = proc.waitFor();
//        System.out.println("ExitCode: "+exitVal);
    } catch (Throwable t) {
        t.printStackTrace();
    }
```

Dieser Code kann an beliebiger Stelle eingebaut werden und ruft dann den Konvertiervorgang auf. In der Variablen **exitVal** befindet sich der Exit-Code des Konverters, anhand dessen dann über die weitere Ausführung der Software entschieden werden kann.

7.2.4.5 Einschränkungen/Besonderheiten bei der TND-Erzeugung

Im Folgenden werden einige Einschränkungen und Besonderheiten dieser Version des DXF/CAD-nach-TND-Konverters beschrieben.

7.2.4.5.1 Slipswitches Diese Version des Konverters unterstützt keine Slipswitches. Bei der Fertigstellung dieser Version wurde aus Zeitgründen auf eine Implementierung dieser doch etwas komplexeren Gleisnetzelemente verzichtet. Dies wird aber nicht als allzu gravierend angesehen, zumal sie in Straßenbahnnetzen doch auch recht selten vorkommen und somit kaum Einschränkungen in der Nutzbarkeit diesbezüglich vorhanden sein sollten.
Es besteht immer noch die Möglichkeit, diese Unterstützung in einer Folgeversion einzubauen, zumal die CAD-Elemente für Slipswitches bereits existieren. Für TRACS wird dies aber nicht mehr erfolgen.

7.2.4.5.2 Markierungen Bei den Markierungen gibt es ebenfalls ein paar Einschränkungen, bzw. restriktivere Regeln. Während die TND keine expliziten Eingangs- und Ausgangsmarkierungen erfordert (Sensoren können diese Aufgabe übernehmen), werden sie bei der Umsetzung eines DXF-Gleisnetzes zwingend benötigt. Da hier nur eine unwesentliche Option der TND eingeschränkt wird, sollte auch dies keine Probleme bei der Erzeugung von Gleisnetzen hervorrufen.
Problematischer könnte hingegen die in dieser Version unzureichende Unterstützung von mehreren Markierungen in Folge sein. In der aktuellen Version ist es nicht erlaubt, mehrere Markierungen hintereinanderzusetzen, ohne dass ein anderes Element dazwischen liegt. Für die Darstellung von Kurven und ähnlichem ist dies nun eine relativ starke Einschränkung.
Da aber TRACS kleine Ausschnitte eines großen Netzplanes als Gleisnetz ansieht, kann man mit dieser Einschränkung hoffentlich leben. In einer zukünftigen Version sollte diese Unzulänglichkeit in jedem Fall beseitigt werden.

Es ist auch zu beachten, dass sich die Markierungen für Eingang, Ausgang und auch für ein totes Gleisende (Sackgasse) untereinander und von den einfachen Markierungen unterscheiden. Es dürfen aber auch keine einfachen und Sondermarkierungen aufeinander folgen.

7.2.4.5.3 Signale Signale werden in dieser Implementation ausschließlich als Markierungen zwischen zwei Sensoren interpretiert. Die TND erlaubt aber darüberhinaus noch die Darstellung als Sensor selbst. Auch dies wird nicht als Einschränkung in diesem Sinne angesehen - eher als restriktivere Regel.

7.2.4.5.4 Test-Version Neben der beschriebenen stabilen Version existiert auch eine Test-Version, in welcher einige Einschränkungen nicht mehr vorhanden sein sollten. So sind zum einen Mehrfach-Markierungen (mehrere Markierungen in Folge) hier möglich. Zum anderen gibt es eine Basis-Unterstützung von Slipswitches. Hierbei ist allerdings anzumerken, dass zur Normalausrichtung verdrehte Slipswitches nicht korrekt konvertiert werden, da die Koordinaten der internen Weichen nicht einwandfrei berechnet werden können.
Da diese Version nicht ausreichend getestet werden konnte, ist sie als Zweit-Version hier vorhanden. Benutzung auf eigene Gefahr!!

7.2.4.6 Abschließende Anmerkungen

Es folgen noch ein paar Anmerkungen, welche beim abschließenden Testen aufgefallen sind.

7.2.4.6.1 Zeichenfehler Da sich bei den Tests herausgestellt hat, dass DXF-Zeichnungen manchmal etwas „eigenwillig“ abgespeichert werden, kann es trotz anscheinender Korrektheit der Zeichnung - jedenfalls, was die Darstellung in QCad betrifft - zu internen Fehlern kommen, welche trotz intensiver Bemühungen nicht endgültig geklärt werden konnten, da sie zumeist nicht reproduzierbar waren. Dies tritt vor allem bei größeren Gleisnetzen auf und äußert sich in fehlerhaften Nachbarschaftsbeziehungen.
Daher ist in dieser Version des Konverters leider dazu zu raten, diese Beziehungen noch einmal anhand der Zeichung manuell zu überprüfen.
Interessanterweise ist durch Löschen des betreffenden Blockes (Element incl. Benennung) und erneutem Einfügen des Elementes, Neubenennung desselben und Erzeugung eines neuen Blockes aus diesen beiden Teilen - dann aber evtl. unter einem anderen Namen, als bei dem zuvor gelöschten (der Name ist für die Weiterverarbeitung und Repräsentation in der TND vollkommen irrelevant) das Problem oftmals behebbar. Diese Beobachtung lässt eher darauf schließen, dass die Speicherung einer umfangreicheren DXF-Datei mittels QCad nicht ganz fehlerfrei zu sein scheint. Was dabei genau passiert

ist aber leider nicht endgültig klärbar gewesen, so dass durchaus auch ein kleiner Fehler innerhalb des Konverters die Ursache sein kann.
Leider kann es dabei auch passieren, dass die Konvertierung als solches abbricht, da intern eine Ausnahmesituation aufgetreten ist, die die weitere Ausführung unmöglich macht. Die Ursache ist dieselbe, hat nur aufgrund der internen Programmstruktur in gewissen Situationen eben diese Auswirkung.

Henrik Röhrup

7.2.5 Reflexion

7.2.5.1 Überblick

In dem folgenen Abschnitt soll die Arbeit in der Netzgraphgruppe allgemein bewertet und die aufgetretenen Probleme aufgezeigt werden.

7.2.5.2 Bevor die Probleme begannen

Die Netzgraphgruppe wurde zum Ende des ersten Projektsemesters ins Leben gerufen, nachdem der Netzgraph überhaupt erst an dem Projektwochenende in jenem Semester auf die Tagesordung gehoben wurde. In der verbleibenden Zeit jenes Semesters wurden vorrangig allgemeine Informationen zur Thematik recherchiert und unter anderem die zu verwendende CAD-Software festgelegt.
Zu diesem Zeitpunkt war die Gruppe aber nur eine Art Nebengruppe, da deren Mitglieder in anderen Gruppen aktiv waren. Erst mit Beginn des zweiten Semesters wurde die Gruppe zu einer vollwertigen Gruppe - mit hauptsächlich neuer Besetzung.

7.2.5.3 Gruppeninterne Probleme

Im zweiten Projekt-Semester gestaltete sich dann die Arbeit in der Netzgraphgruppe als „schwierig“, da zwei Drittel der Gruppenmitglieder vorzeitig ihre Arbeit eingestellt und bis zum Ende des Semesters das Projekt TRACS verlassen haben, so dass letztlich die Arbeit nur von einer Person durchgeführt werden konnte. Da bis zu jenem Zeitpunkt die Kommunikation innerhalb der Gruppe trotz mehrfacher Versuche und Einberufung von Treffen seitens des am Ende verbliebenen Mitglieds niemals richtig funktionierte, ist es folglich auch zu mehreren Situationen gekommen, wo angefangene Arbeit verworfen werden musste, da die entsprechenden Gruppenmitglieder nicht mehr zur Verfügung standen und grundlegende Informationen zum Lösungsansatz anderweitig nicht existierten. Von der ursprünglichen Besetzung der Gruppe aus dem ersten Semester war zudem niemand mehr in der Gruppe vertreten.

7.2.5.4 Zeitnot durch Personalverluste

Das Ergebnis ist folglich nicht das erwartete und sämtliche ursprünglich gemachten Zeitabschätzungen mussten somit auf eine Person umgerechnet werden, was den Zeitrahmen gänzlich ins Wanken brachte.
Einzig die Abschätzung zur „Einarbeitung in das CAD-System“ von 8-16 MW konnte - den Zeitraum von rund eineinhalb Monaten betrachtet - halbwegs eingehalten werden. Die effektiv benötigte Zeit war dabei allerdings deutlich geringer als die geschätzte, was ja aber aufgrund der oben genannten Probleme keinen wirklichen Gewinn darstellte und nach außen hin nicht auffiel.
Die zu Beginn gemachte Abschätzung des Arbeitsaufwandes von 60 MW für den CAD-TND-Konverter wäre vermutlich realistisch gewesen, wenn genannte gruppeninterne Probleme nicht aufgetreten wären. Da die Netzgraphgruppe ab dem dritten Projekt-Semester auch offiziell nur noch mit einer Person weitergearbeitet hat und somit eigentlich nicht mehr als Gruppe bezeichnet werden konnte, musste die Gesamt-Zeiteinschätzung für den Konverter zur Halbzeit des Projektes auf rund 80 MW angehoben werden, da nur die Ideen einer Person als Basis vorhanden waren. Mehrere Personen ergänzen sich in der Regel gegenseitig und es würden sich entsprechend mehr Ideen für Lösungsansätze ergeben. Entsprechend zog sich der Zeitplan um einiges in die Länge. Diese Neu-Einschätzung stimmte dann am Ende aber ganz gut mit der wirklichen Dauer überein.
Eine zwischenzeitlich gewünschte Verstärkung der Gruppe war leider mangels Personals nicht möglich. Es wurde dagegen zu Beginn des **dritten** Projekt-Semesters der Vorschlag gemacht, die Netzgraphgruppe mit der Compilergruppe zusammenzulegen - dies wurde aber niemals endgültig beschlossen, obwohl anderslautende Meinungen diesbezüglich nie wirklich verstummten. (Anmerkung: Es findet sich dazu auch in keinem Plenums-Protokoll ein Hinweis auf einen derartigen Beschluss - lediglich auf diesen Vorschlag - wie auch, die Netzgraphseite hat niemals die Zustimmung dazu gegeben und über den Kopf eines betroffenen Projekt-Mitglieds hinweg kann auch ein Plenum eine solche Entscheidung nicht treffen, da dies dann andere für das gesamte Projekt unangenehme Konsequenzen zur Folge gehabt hätte.)

7.2.5.5 Personalkarussell

Die Idee hinter diesem Vorschlag war es jedenfalls, durch eine Verstärkung der Compilergruppe die Fertigstellung des Compilers zu beschleunigen und außerdem die dort gewonnenen Erfahrungen später im Netzgraphen einfließen zu lassen, was vom Prinzip her sicherlich ganz sinnvoll gewesen wäre.
Da aber bei beiden Gruppen eine Einarbeitung der jeweils anderen Gruppenmitglieder hätte erfolgen müssen, welche aus Sicht der Netzgraphgruppe beim Blick auf Verständnisfragen bei der Aufgabe des Compilers dort auf jeden Fall doch relativ groß gewesen

wäre, hätte man das Ziel einer Gruppenverschmelzung ernsthaft weiter verfolgt, wurde diese Idee recht schnell von der Netzgraphseite verworfen. Auch hätte es für die Netzgraphseite eine gravierende Umstellung hinsichtlich auf die beim Compiler verwendete Programmiersprache gegeben, was ebenfalls zu einer erhöhten Einarbeitungszeit geführt hätte (siehe dazu auch die Entscheidung für die Programmiersprache Java™im Abschnitt 7.2.3.2.1 auf Seite 89).
Im Hinblick auf diesen Tatbestand hätte eine Verschmelzung dieser Art nach Auffassung des einzig verbliebenen Netzgraph-Mitglieds wenig Sinn gemacht, zumal anschließend sehr wahrscheinlich dasselbe Problem bei der Wiederaufnahme an der Netzgraph-Implementation aufgetreten wäre und unterm Strich tendenziell eher ein Zeitverlust als eine Beschleunigung aufgetreten wäre. Von anderer Seite her kam im Übrigen auch nie der ernsthafte Versuch einer Zusammenlegung zustande. Aus der Kommunikation zwischen den einzelnen Gruppen konnte man auch zu absolut keiner Zeit entnehmen, dass durch diesen Entschluss der Netzgraphseite beim Compiler irgendwelche neuen Probleme aufgetreten wären.
Auf der anderen Seite hätte die Compiler-Gruppe nicht in die Zeitnot geraten müssen, in der sie sich am Ende befand, wenn sich deren Mitglieder nicht neben dem Projekt TRACS noch mit einer anderen offenbar sehr zeitaufwändigen Sache beschäftigt hätten - aber dies ist eine andere Sache...
Wie dem auch sei - die gewonnene Erkenntnis auf der Netzgraphseite bei dieser Angelegenheit war jedenfalls die, dass das Einarbeiten neuer Leute in ein bestehendes (Teil-)Projekt viel Zeit in Anspruch nehmen kann, welche vielleicht besser in Lösung der eigentlichen Aufgabe gesteckt werden sollte. Eine Aufstockung der Gruppenstärke sollte immer rechtzeitig erfolgen und nicht erst, wenn die Arbeit im vollem Gange ist, da diese vermeintliche Verstärkung letzlich wenig bringt und unter Zeitdruck eher dazu führt, dass sich die Fertigstellung des gesamten Projektes eher noch mehr verzögert.

Da zwischenzeitlich auch in anderen Gruppen Personalprobleme aufgetreten waren und der Gesamt-Zeitplan daher als immer kritischer anzusehen war, je weiter die Zeit voranschritt, wäre es auch fraglich gewesen, ob nach Ende der Compilerarbeit wirklich eine vereinte Netzgraph-Compiler-Gruppe in dieser Form die Arbeit am Netzgraphen wieder aufgenommen hätte. Wahrscheinlicher wäre es gewesen, dass allenfalls zwei Personen am Netzgraphen weitergearbeitet hätten und der Netzgraph-Konverter nicht in dieser Form zustandegekommen wäre, was unter Umständen auch den TND-Builder in Frage gestellt hätte, da dieser vom Netzgraph-CAD-TND-Konverter mehr oder weniger abhängig ist.

7.2.5.6 Andere Probleme

Aber auch an anderer Stelle wurde zu viel Zeit verloren: Die Frage bezüglich der möglichen Verwendung von *CUP Parser Generator for Java™* hat ebenfalls einige Zeit gekostet, so dass beides im Hinblick auf eine Motivationssteigerung eher hinderlich war

und unterm Strich leider rund 6 Wochen Zeit verloren gegangen sind. Die Diskussion über das Für und Wider ist in Abschnitt 7.2.3.2.2 auf Seite 89 zu finden.
Darüberhinaus gab es noch einige andere Probleme, welche darauf zurückzuführen sind, dass eine Person alleine bei einer Planung mehr Fehler macht oder Probleme übersieht, als wenn mehrere Personen sich der Sache annehmen.
So wurde zum Beispiel das Erkennen der Nachbarschaftsbeziehungen vollkommen falsch eingeschätzt, woraufhin letztlich vier Wochen an Fehlersuche eingeschoben werden mussten, so dass gegen Ende ein gewisser Zeitdruck entstand, welcher aber noch im Rahmen blieb, da von vorn herein mit einem gewissen Zeitpuffer geplant wurde.
Wiederholte Änderungen an der TND-Beschreibung waren auch nicht unbedingt hilfreich, aber die daraus resultierenden Änderungen am Konverter hielten sich glücklicherweise in Grenzen.
Letztlich ist ein gut funktionierender DXF-nach-TND-Konverter entstanden, welcher sicherlich besser hätte sein können, vor allem, was die Fehlerbehandlung betrifft, aber die eigentlich gewünschte Funktionalität ist vorhanden.
Kurz vor Abgabeschluss wurden sogar noch ein paar Funktions-Einschränkungen behoben, welche allerdings nicht mehr ausführlich getestet werden konnten.

7.2.5.7 Gruppenübergreifende Zusammenarbeit

Zur Zusammenarbeit mit anderen Teilgruppen ist zu sagen, dass es eine direkte Zusammenarbeit so nicht gab, da dies nicht notwendig war, da der Netzgraph-Konverter ein eigenständiges Programm darstellt, welches keine direkten Schnittstellen zu anderen Teilen des Projektes hat. Es wird lediglich das Produkt einer Konvertierung von anderen Teilgruppen - insbesondere von der Komponente *TND-Builder* - weiterverarbeitet. Das Format des Produktes - nämlich die TND - ist aber von der DSL-Gruppe bereits definiert worden. Hierbei gab es allerdings mit der Zeit ständige Änderungen, sodass hier mehr oder weniger eine indirekte Zusammenarbeit mit den entsprechenden Gruppen vorhanden war.

7.2.5.8 Alternativer Lösungsansatz

Abschließend soll eine alternative Lösungsidee für die Aufgabe der Erfassung eines Gleisnetzes und Umsetzung in die TND nicht unerwähnt bleiben.
Diese Idee kam im Arbeitsbereich Netzgraph auf, als immer klarer wurde, welche Fehlerquellen von Eingabe der Zeichnung bis zur Konvertierung entstehen können. Zu diesem Zeitpunkt war es aber nicht mehr möglich auf diesen Lösungsweg umzuschwenken, weil zum einen die Zeit nicht mehr ausreichen würde und zum anderen die Arbeit von fast zweieinhalb Semestern umsonst gewesen wäre und quasi nur noch als „Programmierübung“ da stehen würde. Der Frust wäre dann sicherlich noch viel größer gewesen, als er es aufgrund der schwierigen Bedingungen so schon war. Wenn man während des

Projektwochenendes detaillierter über andere Lösungswege nachgedacht hätte und sich nicht so schnell auf die nun vorliegende CAD-Variante fixiert hätte, so wäre man vielleicht schon zu jenem Zeitpunkt auf etwas ähnliches gekommen.
Aber nun zur eigentlichen Idee:

7.2.5.8.1 Grafisches Baukastensystem Anstatt die Eingabe des Gleisnetzes mittels einer CAD-Software vorzunehmen, um deren Ausgabedatei dann zu konvertieren, hätte man gleich eine Eingabemöglichkeit wählen sollen, welche eine Konvertierung überflüssig macht.
Ein Baukastensystem mit grafischer Benutzeroberfläche, in welchem man eine - wie im CAD-Programm - vordefinierte Element-Bibliothek zur Verfügung hätte, mit dessen Hilfe man das Gleisnetz direkt zusammensetzen könnte, wäre durchaus die bessere Lösung gewesen.
Die Objekte, welche die Gleisnetzelemente darstellen, wären intern bereits so strukturiert, so dass man durch Zusammensetzen mehrerer Objekte bereits die Nachbarschaftsbeziehungen auslesen könnte. Der ganze umständliche Umrechungs- und Extrahiervorgang würde wegfallen.
Auch die Benennung der Elemente wäre einfacher, wenn man einfach nur das Element markieren und ihm einen Namen geben könnte, welcher dann einfach im Objekt abgespeichert werden würde, anstatt durch zusätzlicher Erzeugung eines eigenen Blockes die Zusammengehörigkeit von Name und Element zu gewährleisten, was auch sehr fehleranfällig sein kann, wenn man nicht ständig genau aufpasst.
Gedrehte Elemente sind ebenfalls einfacher darstellbar - vor allem die Koordinaten-Neuberechungen und Erkennung von Nachbarschaftsbeziehungen sind in der CAD-Lösung in diesem Szenario alles andere als simpel.
Es würde also der Großteil an Fehlerquellen wegfallen. Außerdem wäre ein solches Programm sicherlich einfacher zu implementieren gewesen, wobei hier sicherlich die grafische Oberfläche und die Objektbibliothek einen großen Teil der Zeit beansprucht hätten.
Ebenso würde das zuletzt *TND-Builder* (siehe auch Abschnitt 7.3 auf Seite 123) genannte Arbeitspaket komplett wegfallen, da es in der beschriebenen Lösung recht einfach gewesen wäre, auch die Routen- und Hardware-Klassen-Informationen einzubauen.
So könnte es zum Beispiel eine Funktion geben, in welcher man alle Elemente einer Route hintereinander anklickt und schon wäre die Route als solches komplett. Eine intelligente Methode wäre ebenfalls denkbar, in der einfach nur Start- und Endpunkt und ggf. wichtige Knotenpunkte (für den Fall, dass es mehrere Wege von A nach B gibt) markiert werden und so die Route automatisch anhand der internen Daten berechnet wird. Für die Hardware-Klassen wäre eine ähnliche Funktionalität denkbar, in der einfach alle zu einer Klasse gehörenden Elemente markiert werden.
Die Folge wäre ein Programm gewesen, welches alle Schritte von der Eingabe, bis hin zur TND-Erzeugung durchführt, statt wie in der gewählten Lösung drei Programme zu

haben, von denen eines (QCad) außerdem noch nicht einmal von TRACS stammt.
Eventuell wäre sogar eine Einbindung des Simulators an dieser Stelle möglich gewesen. Jedenfalls hätte man somit auch die Visualisierung von einem als TND-Datei vorliegenden Gleisnetz erhalten können.
Vielleicht ist diese Idee in einem anderen Zusammenhang ja noch verwertbar, aber für TRACS kommt sie leider zu spät.

7.2.5.9 Fazit

Die Arbeit in der Netzgraphgruppe war eher von Frust und Demotivierung geprägt, als durch das, was eigentlich ein Hauptziel in einem Hauptstudiumsprojekt ist - Teamarbeit - denn diese gab es eigentlich nie.
Das Positive ist aber, dass am Ende ein Produkt steht, welches zwar mit kleinen nicht unbedingt gravierenden Einschränkungen behaftet, aber in dem geplanten Rahmen dennoch funktionsfähig ist.
Ein Teil der ursprünglich geplanten Software ist aber letzten Endes entfallen, wobei im Hinblick auf die Probleme auch niemals alles hätte entwickelt werden können.
Da zu Beginn der Netzgraph-Arbeit alles zu schnell auf eine Idee hin ausgelegt war, ist ein besseres Produkt leider verhindert worden.
Dennoch: Die gewonnen Erfahrungen werden für die Zukunft sicherlich hilfreich sein, derartigen Problemen rechtzeitig aus dem Wege zu gehen.

Henrik Röhrup

7.3 TND-Builder

7.3.1 Überblick

Im Rahmen des Projektes wurde ein GUI Werkzeug erstellt, damit der Benutzer die Routen auf einfache Weise definieren kann. Die definierten Routen werden dann in der TND gespeichert. So wird eine endgültige TND-Datei erstellt, die der Compiler weiterverarbeiten kann. Dieses Softwarewerkzeug bezeichnen wir als TND-Builder. Der TND-Builder hat folgende Eigenschaften:

- Er ist für unterschiedliche Gleisnetze wiederverwendbar.
- Durch Intergration des CAD-TND-Konverters kann die CAD-Zeichendatei direkt in die TND konvertiert und weiter bearbeitet werden.
- Bei ungültigen Routendefinitionen erfolgt eine Fehlermeldung.
- Der TND-Builder ist für unterschiedliche Platformen verwendbar. Die Software ist in Java implementiert.

Deng Zhou

7.3.2 Funktionsbeschreibung

7.3.2.1 Ein- und Ausgabe

Der TND-Builder hat zwei Eingabemöglichkeiten. Die erste Möglichkeit besteht darin, eine CAD-Datei einzulesen und in das TND-Format umzuwandeln. Die CAD-Datei soll dabei in QCad erstellt werden und verwendet unsere Zeichen-Bibliothek. Die zweite Möglichkeit besteht darin, eine bereits existierende TND-Datei direkt einzulesen.
Die Ausgabe des TND-Builders ist eine vollständige TND mit Routendefinition. Diese TND-Datei ist für den Compiler direkt verwendbar.

7.3.2.2 Bestandteile einer TND-Datei

Eine TND-Datei besteht aus zwei Teilen.

- Der erste Teil beinhaltet alle Hardwareelemente, die sich im Gleisnetz befinden, die Eigenschaften der Hardwareelemente sowie geographische Beziehungen zwischen den Hardwareelementen.
- Der zweite Teil beinhaltet die Routendefinition für das Gleisnetz sowie die Informationen über Konflikte zwischen Routen.

Ähnlich arbeitet auch der TND-Builder in zwei Phasen.

- In der ersten Phase wird der CAD-TND-Konverter aufgerufen, um den ersten Teil der TND zu generieren.

  Dabei werden definitions-Block, coordinates-Block, signal-properties-Block, point-properties-Block, relations-Block, signal-positions-Block und hardwaremap-Block generiert. Sonstige Blöcke in der TND sind leer(nur Blockname ohne Inhalt).

- In der zweiten Phase können Routendefinitionen erstellt oder bearbeitet werden und Routenkonflikte identifiziert werden.

  In dieser Phase werden routedefinitions-Block, conditions-Block, clearances-Block, conflicts-Block und point-conflicts-Block generiert.

7.3.2.3 Erzeugen einer neuen TND aus einer CAD-Datei

Die Abbildung 7.20 zeigt in einem Screenshot das Erzeugen einer TND-Datei aus einer CAD-Datei. Bevor die Konvertierung stattfindet, muss man eine vorhandene DXF-Datei selektieren und ein TND-Datei als Ziel angeben sowie eine Konfigdatei für den DXF2TND Konverter.

Abbildung 7.20: Erzeugen einer neuen TND

7.3.2.4 Laden einer vorhandenen TND

Die Abbildung 7.21 auf der nächsten Seite zeigt in einem Screenshot das Laden einer vorhandenen TND-Datei.

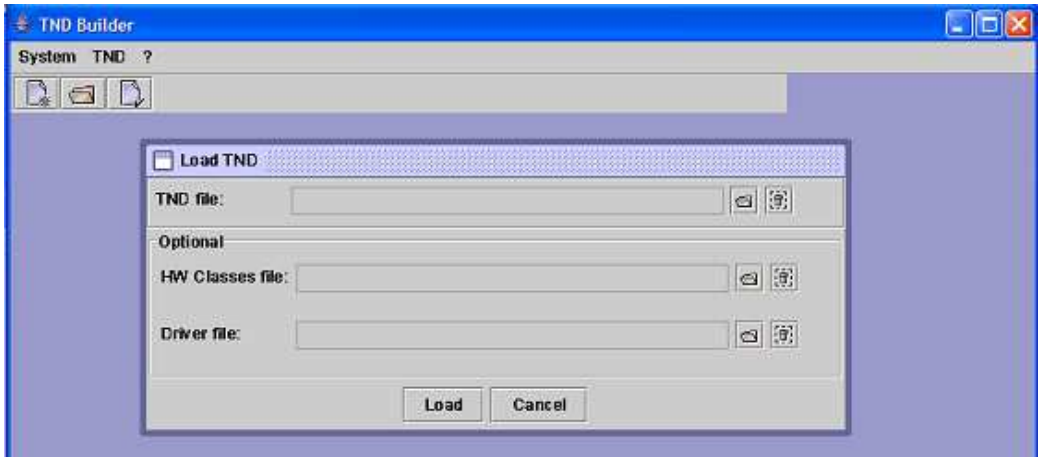


Abbildung 7.21: Vorhandene TND-Datei laden

7.3.2.5 Eigenschaften der Hardwareelemente

Die Abbildung 7.22 zeigt die Darstellung der Eigenschaften der Hardwareelemente. Die Eigenschaften der Hardwareelemente kommen aus der CAD-Datei. Im TND-Builder darf der Benutzer die Eigenschaften nicht ändern.

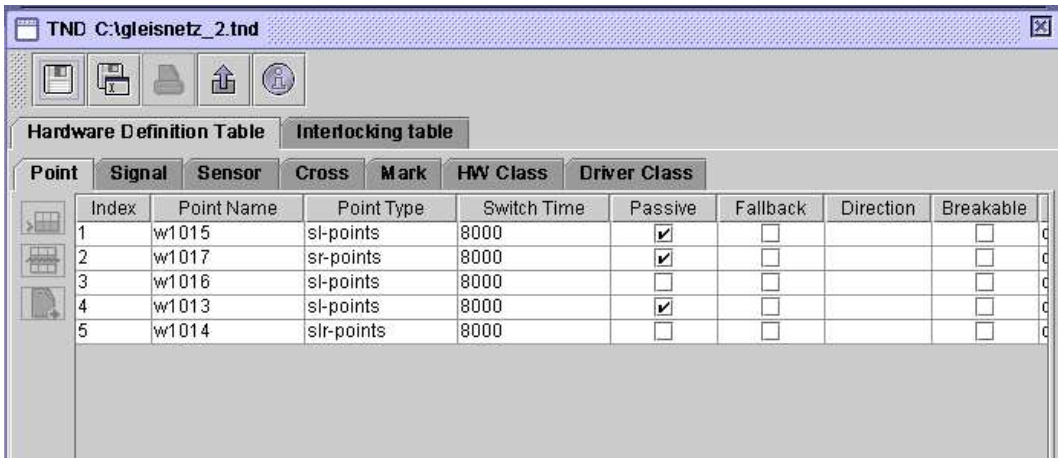


Abbildung 7.22: Hardwareliste und Eigenschaften

7.3.2.6 Routendefinitionen

Die Abbildung 7.23 auf der nächsten Seite zeigt, wie die in der TND-Datei definierten Routen aussehen. Die Darstellung erfolgt gemäß [HP02].

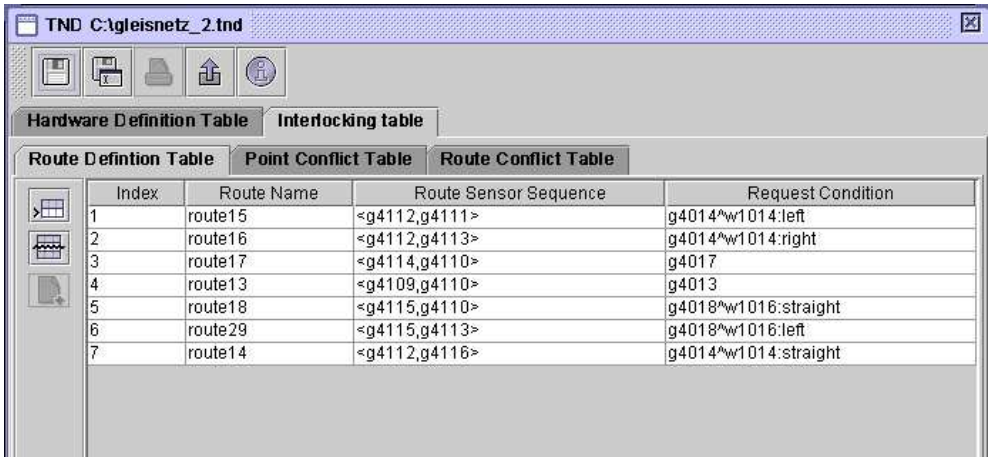


Abbildung 7.23: Definierte Routen

7.3.2.7 Routeninformationen

Wenn man auf eine Route doppelklickt, wird ein Dialog angezeigt, der detaillierte Informationen über die jeweilige Route enthält. Diese Informationen beinhalten beispielsweise eine Liste der Weichen in der Reihenfolge, in der diese befahren werden sowie die jeweiligen Weichenstellungen. Dies ist in Abbildung 7.24 zu sehen.

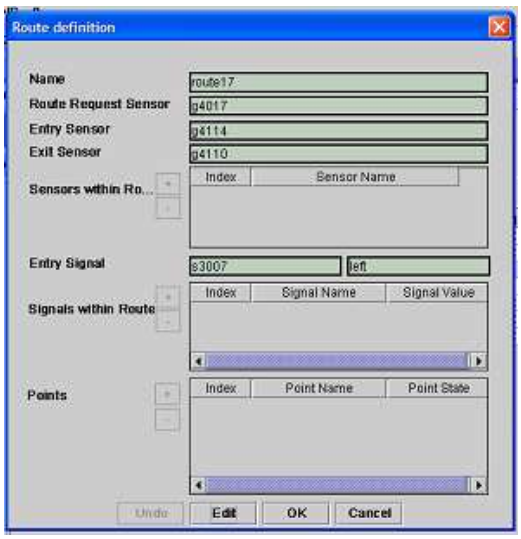
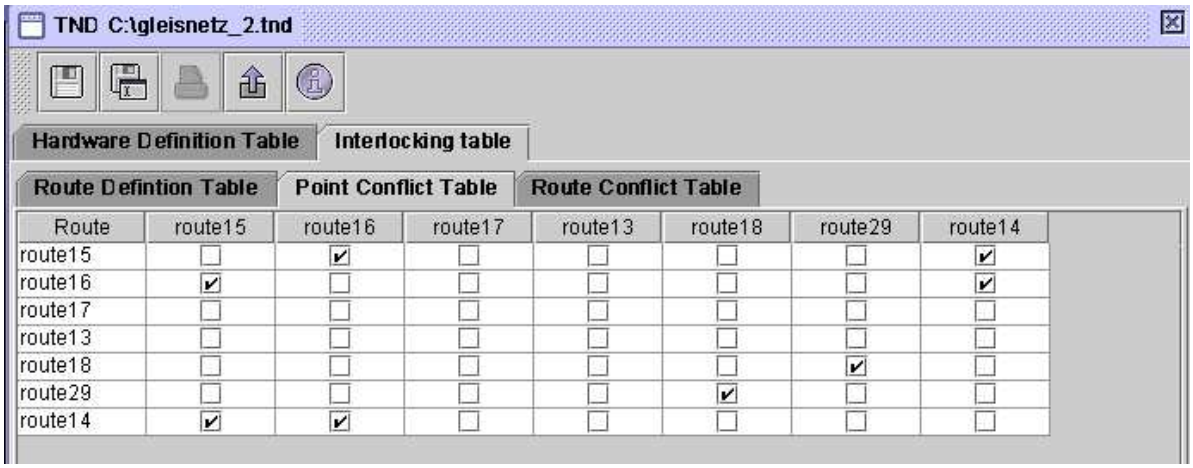


Abbildung 7.24: Routeninformationen

7.3.2.8 Weichenkonflikttabelle

Anhand der Routendefinitionen wird die Weichenkonflikttabelle automatisch generiert. Wenn zwei Routen die gleiche Weiche befahren sollen, besteht ein Weichenkonflikt zwischen diesen beiden Routen. Eine ausführliche Erklärung dazu ist in der Beschreibung des point-conflicts-Blocks im Abschnitt 7.1.3.2.12 auf Seite 75 zu finden. Die Abbildung 7.25 zeigt ein Beispiel einer Weichenkonflikttabelle.



Route	route15	route16	route17	route13	route18	route29	route14
route15	☐	☑	☐	☐	☐	☐	☑
route16	☑	☐	☐	☐	☐	☐	☑
route17	☐	☐	☐	☐	☐	☐	☐
route13	☐	☐	☐	☐	☐	☐	☐
route18	☐	☐	☐	☐	☐	☑	☐
route29	☐	☐	☐	☐	☑	☐	☐
route14	☑	☑	☐	☐	☐	☐	☐

Abbildung 7.25: Weichenkonflikttabelle

7.3.2.9 Routenkonflikttabelle

Zusätzlich zu der automatisch generierten Weichenkonflikttabelle muss der Nutzer auch alle Routenkonflikte in einem Gleisnetz identifizieren, die zu Kollisionen zwischen Bahnen führen können. Diese Konflikte zwischen je zwei Routen werden in der Tabelle durch ein Kreuz markiert. Eine ausführliche Erklärung dazu ist in der Beschreibung des conflicts-Blocks im Abschnitt 7.1.3.2.13 auf Seite 76 zu finden. Die Abbildung 7.26 auf der nächsten Seite zeigt ein Beispiel einer Routenkonflikttabelle.

7.3.2.10 Einschränkungen bei Routendefinitionen

Ein wichtiger Teil des TND-Builders sind die Routendefinitionen. Bei einer Routendefinition muss der Benutzer folgende Punkte beachten:

- In einem Gleisnetz dürfen nicht zwei Routen denselben Namen haben und jeder Routenname muss mit „r“ beginnen.
- Jede Route muss einen Request Sensor haben.

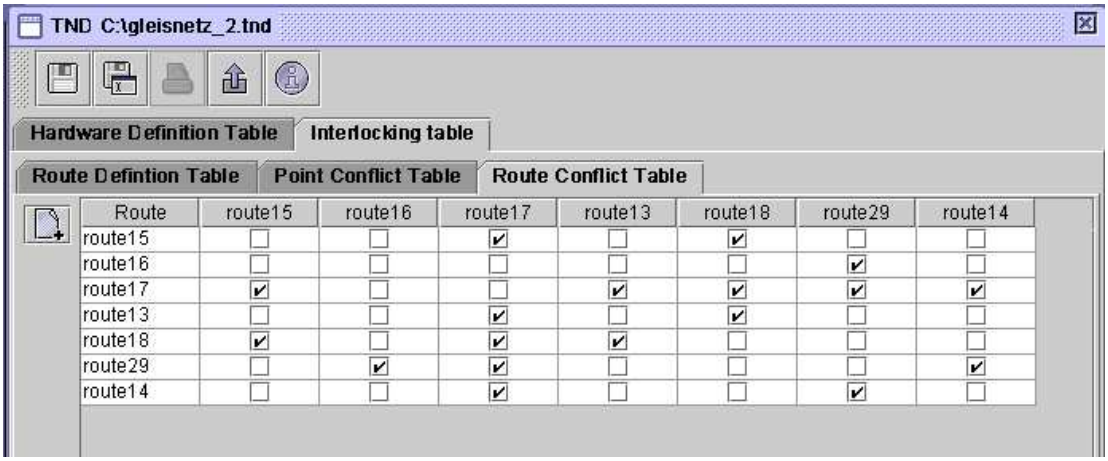


Abbildung 7.26: Routenkonflikttabelle

- Jede Route muss ein Eingangssignal haben und die Go-Richtung muss festgelegt werden.
- Jede Route muss einen Eingangssensor und einen Ausgangssensor haben.
- Für jede Route müssen die für die Route benötigten Weichenstellungen definiert werden. Dabei müssen die Weichenstellungen in der Reihenfolge definiert werden, in der die Weichen befahren werden. Zudem werden nur Weichen berücksichtigt, die nicht passiv sind.
- Passive Weichen und Kreuzungen tauchen nicht in der Routendefinition auf.
- Wenn der Request Sensor und das Eingangssignal nicht zusammenpassen, wird eine Fehlermeldung angezeigt.
- Wenn Request Sensor, Eingangssensor, die zu der Route gehörenden Weichen und der Ausgangssensor nicht in der richtigen Reihenfolge befahren werden können, wird eine Fehlermeldung angezeigt.

Hier wird ein Beispiel für korrekte Nachbarschaftbeziehungen angegeben. Das betreffende Gleisnetz ist in Abbildung 7.27 auf der nächsten Seite zu sehen.

Zum Beispiel hat eine Route als Eingangssensor den Sensor 4101 und als Ausgangssensor 4105.
Falls das Eingangssignal dieser Route nicht 3001 ist, wird eine Fehlermeldung angezeigt, dass Eingangssignal und Eingangssensor nichts übereinstimmen.
Falls der Route Request Sensor nicht 4001 ist, wird eine Fehlermeldung angezeigt, dass Route Request Sensor und Eingangssensor nichts übereinstimmen.

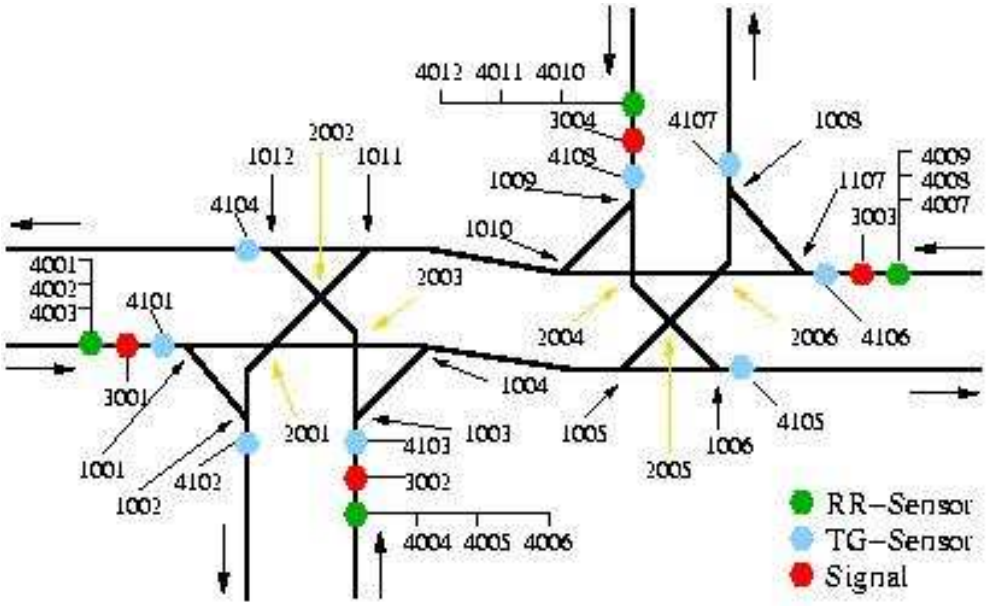


Abbildung 7.27: Beispielgleisnetz

Falls die Weiche 1001 für die betreffende Route nicht geradeaus gestellt ist, wird eine Fehlermeldung angezeigt, dass die Einstellung der Weiche nicht zu dem Ausgangssensor der Route führt.

Deng Zhou

7.3.3 Installation

7.3.3.1 Hardwareanforderungen

Das Programm benötigt einen IBM kompatiblen PC mit mindestens 32M RAM.

7.3.3.2 Softwareanforderungen

Das Programm braucht JDK/JRE (1.4 oder höhere Version).

7.3.3.3 Package

Wir liefern den TND-Builder sowohl als vorkompiliertes Binärprogramm (JAR-Datei) wie auch als Quellcode.
Wer das Programm direkt ausführen möchte, braucht nur diese JAR-Datei zum JAVACLASSPATH hinzuzufügen und das Programm mit folgendem Befehl zu starten:

- java -jar tndbuilder.jar

Für diejenigen, die das Programm selber kompilieren wollen, haben wir ein Makefile für diesen Zweck erstellt. Man braucht dazu das Makefile einmal mit fogendem Befehl aufzurufen, wodurch eine endgültige JAR-Datei erstellt wird:

- make release

Deng Zhou

7.3.4 Implementierung

7.3.4.1 Programmierssprache

Als Programmierssprache für den TND-Builder verwenden wir Java. Weil der TND-Builder ein User-Interface Programm ist, um Routen in einem Gleisnetz zu definieren, sollte er nicht mit einem speziellen System oder Betriebssystem verbunden sein, sondern in möglichst verschiedenen Systemen einsetzbar sein. An dieser Stelle ist Java die beste Wahl für uns.

7.3.4.2 Programmstruktur

Die Programmstruktur gibt einen Überblick über die Komponenten, aus denen der TND-Builder besteht. Dies sind insgesamt 4 Komponenten.

- DXF2TND-Konverter (externer CAD-TND-Konverter)
- TND-Parser
- Objekt-Darstellung
- GUI

7.3.4.2.1 DXF2TND-Konverter Die Komponente „DXF2TND-Konverter“ wird von der Netzgraph-Gruppe übernommen. Das Programm wird hier einfach aufgerufen, um aus der eingegebenen DXF-Datei eine TND-Datei zu erzeugen. Eine ausführliche Erklärung dazu ist im Abschnitt 7.2 auf Seite 82 zu finden.

7.3.4.2.2 TND-Parser Die Komponente „TND-Parser“ wird von der Simulator-Gruppe übernommen. Diese hat bereits eine Komponente BestTramNetworkBuilder-OfTheUNIverse erstellt. Diese Komponente liest eine TND-Datei ein und prüft, ob die Datei lexikalisch und syntaktisch korrekt ist. Am Ende erzeugt sie eine Objektdarstellung für das betreffende Gleisnetz. Eine ausführliche Erklärung dazu ist in der Beschreibung der Simulator-Programmstruktur in Abschnitt 7.8.4.2 auf Seite 328 zu finden.

7.3.4.2.3 Objekt-Darstellung Die Komponente „Objekt-Darstellung“ modelliert alle Objekte, die für die Routendefinitionen wichtig sind. Zudem führt sie die Umwandung von Routenobjekten zu Routenblöcken aus.
Die Abbildung 7.28 zeigt das Klassendiagramm für TND-Objekte.

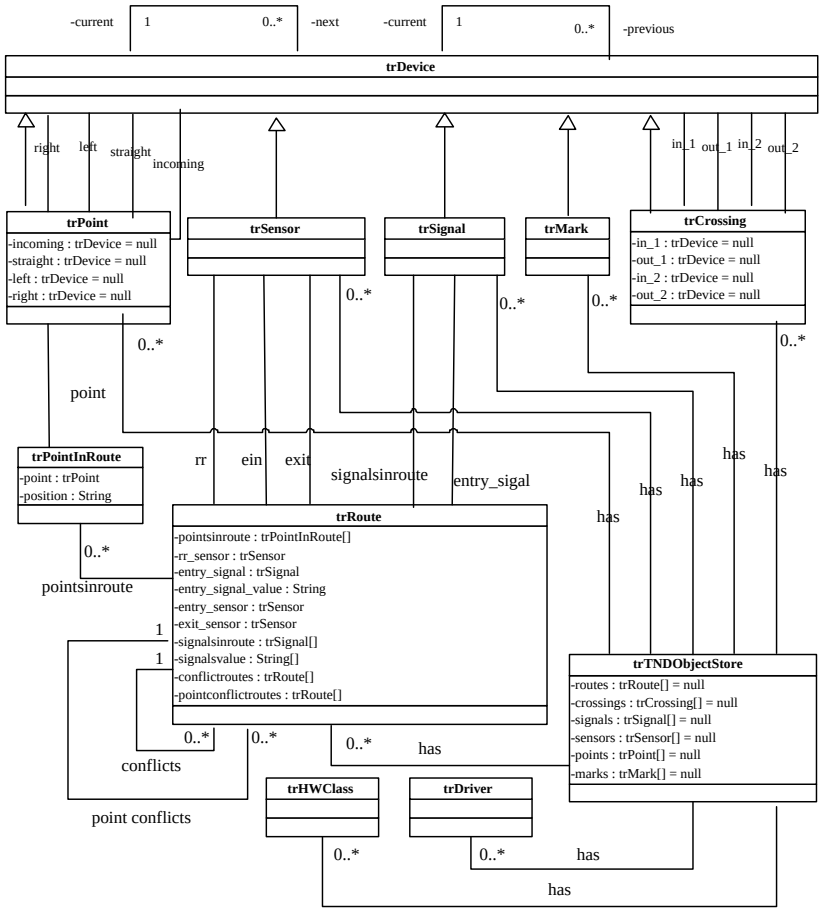


Abbildung 7.28: Klassendiagramm

Hier werden alle wichtigen Klassen kurz erklärt.

- trDevice „trDevice“ ist die Oberklasse. Sie ist eine Abstraktion der Hardwareelemente, die im Gleisnetz existieren. Zwischen den einzelnen Hardwareelementen gibt es direkte Nachbarschaftsbeziehungen. Die Nachbarschaftsbeziehungen wird durch die Reihenfolge festgelegt, in der sie von Bahnen befahren werden können. Jedem Hardwareelement ist eine Liste von direkt vorhergehenden Hardwareelementen und eine Liste von direkt nachfolgenden Hardwareelementen zugeordnet.

- trPoint, trSignal, trSensor, trMark, trCrossing Dies sind alle konkreten Hardwaretypen, die in der TND vorkommen. „trPoint“ repräsentiert die im Gleisnetz

enthaltenen Weichen. Ein Objekt von diesem Typ kann mit maximal vier anderen Hardwareelementen verbunden werden. „trCrossing“ repräentiert die im Gleisnetz enthaltenen Kreuzungen. Ein Objekt von diesem Typ kann mit maximal vier anderen Hardwareelementen verbunden werden.Die Nachbarschaftssituationen für „trPoint“ und „trCrossing“ sind anderes als für „trDevice“. Bei „trPoint“ ist das Hardwareelement, das sich auf der Seite incoming befindet, benachbart mit den Hardwareelementen auf den Seiten straight, left oder right. Die Hardwareelemente auf der Seite straight, left bzw. right ist nur benachbart mit dem Hardwareelement auf der Seite incoming. Bei „trCrossing“ ist das Hardwareelement auf der Seite in1 nur benachbart mit dem Hardwareelement auf der Seite out1 und das Hardwareelement auf der Seite in2 ist nur benachbart mit dem Hardwareelement auf der Seite out2.

- trRoute „trRoute“ repräsentiert die für das jeweilige Gleisnetz definierten Routen. Sie ist eine wichtige Klasse im Klassendiagramm. Die Klasse hat folgende Attribute:
  - einen Route Request Sensor
  - einen Eingangssensor
  - einen Ausgangssensor
  - ein Eingangssignal
  - eine Liste von Signale, die sich auf der Route befinden, und ihre GO-Richtungen
  - eine Liste von Weichen, die sich auf der Route befinden, und ihre Einstellungen
  - eine Liste von Konfliktrouten
  - eine Liste von Routen, zu denen Weichenkonflikte bestehen
- trTNDObjectStore „trTNDObjectStore“ umfasst alle für ein Gleisnetz benötigten Objekte. So sind alle Hardwareelemente und alle logischen Routen in „trTNDObjectStore“ gespeichert.

7.3.4.2.4 GUI Die Komponente „GUI“ ist für die graphische Darstellung von Komponente 3 verantwortlich. Dadurch kann der Benutzer die Routen einfach und bequem definieren und die Eingabe wird visualisiert.
Hier werden alle wichtigen GUI-Komponenten kurz erklärt.

- trMainFrame

  „trMainFrame“ ist das Hauptfenster des Programms und enthält eine Toolbar und ein Menue. Er kann mehrere trTNDInternalFrame gleichzeitig beinhalten.

- trTNDLoadFrame

  „trTNDLoadFrame“ sammelt Informationen, die zum Einlesen der TND benötigt werden. Dazu muss eine TND-Datei angegeben werden. Die Hardware Class Datei und die Driver Class Datei sind optional. Das Format von Hardware Class Datei und Driver Class Datei ist im Abschnitt 7.5.4.7 auf Seite 199 zu finden

- trTNDInternalFrame

  „trTNDInternalFrame“ ist für eine TND-Datei zuständig. Er beinhaltet mehrere Tabpanels. Jedes Tabpanel zeigt einen Teil der TND-Datei und kann in folgende Kategorien eingeteilt werden:

  - Hardware Definition Tab
    - Weichen Tab
    - Signale Tab
    - Sensoren Tab
    - Cross Tab
    - Mark Tab
    - HW Class Tab
    - Driver Class Tab
  - Verschlusstabellen Tab
    - Route Definition Tab
    - Weichen Konflikt Tab
    - Routen Konflikt Tab

- trRouteDialog

  „trRouteDialog“ ist ein Dialog, der für Routendefinitionen und Routendarstellungen zuständig ist.

Deng Zhou

7.3.5 Reflexion

In diesem Abschnitt wird zunächst dargestellt, was in den einzelnen Semestern erreicht wurde und dann erfolgt ein Ausblick auf die weitere Entwicklung des TND-Builders.

7.3.5.1 Verlauf in den einzelnen Semestern

Mit der Arbeit am TND-Builder wurde erst im vierten Projektsemester begonnen. Der Grund dafür war, dass ein graphisches Werkzeug für Routendefinitionen entstehen sollte. Bei der Entwicklung des TND-Builders wurden teilweise Komponenten von anderen Gruppen übernommen um den Entwicklungsaufwand zu reduzieren, z.B. der DXF2TND Konverter und der TND-Parser. Aufgrund von Anmerkung im Plenum wird im TND-Builder auch die Gültigkeit der Routen geprüft.

7.3.5.2 Weitere Entwicklung des TND-Builders

Der TND Builder kann weiter entwickelt werden. So könnte er eine zentrale Rolle im Entwicklungsprozess spielen. Der Benutzer könnte im TND-Builder den Compiler aufrufen, um die Projektierungsdaten für das jeweilige Gleisnetz zu erzeugen. Danach könnte durch Integration mit der generischen Steuerungssoftware und der Treiber-Bibliothek die gesamte Steuerungssoftware generiert werden. Das ist der einfachste und bequemste Weg ein Gleisnetz-Steuerungssystem für den Kunden zu generieren.

Deng Zhou

7.4 Compiler

Der TRACS Compiler *tram* dient zur Übersetzung von Gleisnetzinformationen in binäre Projektierungsdaten, welche vom TRACS Steuerinterpreter eingelesen und benutzt werden können, um ein entsprechendes Gleisnetz sicher zu steuern. Da der Compiler dem TRACS Steuerinterpreter seine 'Munition' zum Steuern von Gleisnetzen liefert, entstand der Name *tram* aus '**TR**ACS **Am**munition'.

7.4.1 Überblick

Gleisnetzinformationen werden in Form einer textuellen Beschreibungssprache vorgegeben, welche alle zur Steuerung relevanten Daten über ein Gleisnetz enthält. Diese Informationen werden vom *tram* Compiler eingelesen und verarbeitet.
Als Resultat entstehen Binärdaten,welche einem vom TRACS Steuerinterpreter vorgegebenen Format entsprechen. Diese binären Eingabedaten für den Steuerinterpreter sind so strukturiert, dass sie effizient zum Steuern des ursprünglich beschriebenen Gleisnetzes genutzt werden können.
Der folgende Abschnitt erläutert detailliert, welche Anforderungen an den *tram* Compiler gestellt werden, wie er aufgebaut ist, und auf welche Weise er seine Aufgaben erfüllt. Zusätzlich werden externe sowie die wichtigsten internen Schnittstellen beschrieben.
Abschnitt 7.4.2 beschreibt die Aufgaben des *tram* Compilers mit Bezug auf diejenigen TRACS Komponenten, die vom *tram* Compiler abhängen. Dieser Abschnitt dient hauptsächlich der Einordunug des Compilers in den Gesamtrahmen des TRACS Projektes.
Abschnitt 7.4.3 beschreibt den Aufbau des *tram* Compilers. Hierbei werden die einzelnen Stufen der Generierung von binären Projektierungsdaten für den TRACS Steuerinterpreter entsprechenden Teilen des *tram* Compilers zugeordnet.
Abschnitt 7.4.4 erläutert das Vorgehen des *tram* Compilers bei der Erstellung von binären Projektierungsdaten. Jeder Zwischenschritt der Verarbeitung wird hierbei soweit vorgestellt, dass ein Verständnis des Gesamtablaufs möglich ist.
Abschnitt 7.4.5 gibt Einblick in sowohl die externen Schnittstellen des *tram* Compilers, als auch die wichtigsten internen Schnittstellen, die zur Generierung von binären Projektierungsdaten benutzt werden.
Abschnitt 7.4.6 beschreibt schliesslich, welche Systemvoraussetzungen gelten, um den TRACS *tram* Compiler benutzen zu können.

Helge Löding

7.4.2 Anforderungen

In diesem Abschnitt werden die Anforderungen an den TRACS *tram* Compiler beschrieben. Zunächst werden die notwendigen Eingabedaten für den Compiler beleuchtet. Hier

wird darauf eingegangen, welche Informationen welche Relevanz für die zu erstellenden binären Projektierungsdaten haben.
Darauf folgt eine Beschreibung der Ausgaben des *tram* Compilers, in der beleuchtet wird, welche Bedingungen die zu generierenden Projektierungsdaten in Bezug auf die Eingabedaten erfüllen müssen.
Abschliessend folgt eine Einordunung des Compilers in den Verifikationsprozess des Gesamtsystems. Hierbei wird betrachtet, welche Anforderungen der *tram* Compiler in Bezug auf die Sicherheit des resultierenden Steuerungssystems erfüllen muß.

7.4.2.1 Eingabedaten

Die Eingabedaten des Compilers bildet eine Textdatei, welche alle zur Steuerung eines Gleisnetzes notwendigen Informationen enthält. Diese Textdatei muss hierbei der vom Projekt TRACS entwickelten domänenspezifischen Beschreibungssprache *TND* oder *Tram Network Description* entsprechen.
Die TRACS *TND* enthält hierbei drei verschiedene Typen von Informationen, aus denen verschiedene Teile der resultierenden Projektierungsdaten generiert werden. Zu unterscheiden sind Informationen über Gleisnetzelemente, Informationen über einzelne Routen und die dabei involvierten Gleisnetzelemente, und Konflikte zwischen einzelnen Routen.

7.4.2.1.1 Elementinformationen Die Beschreibung einzelner Gleisnetzelemente beinhaltet unter Anderem Informationen über den Typ eines Elementes, dessen Eigenschaften, sowie zu benutzende Treiber. Diese Informationen werden benötigt, um dem TRACS Steuerinterpreter binäre Vorgaben über Steuerimpulse, sowie Sicherheitsbedingungen für das entsprechende Gleisnetzelement machen zu können. So gehen unter Anderem im folgenden Beispiel aus den Informationen für ein Signal `s3001` innerhalb der resultierenden Projektierungsdaten binäre Zeitbedingungen und Informationen über Treiberzugehörigkeit hervor:

```
definitions {
   ...
   sr-signals: s3001, s3002, s3003, s3004;
   ...
}

signal-properties {
   ...
   s3001: wait-straight wait-right rr-straight rr-right switchtime 1000;
   ...
}
```

```
hardwaremap {
   ...
   c1: s3001, s3002, s3003, s3004;
   ...
}
```

7.4.2.1.2 Routeninformationen In der TRACS *TND* finden sich benötigte Informationen über Zugehörigkeiten von Gleisnetzelementen zu definierten Routen. Hier finden sich etwa Vorgaben über Weichenstellungen für bestimmte Routen, oder Angaben über Signalstellungen für das Freigeben oder Sperren von einzelnen Routen. Diese Informationen werden innerhalb des *tram* Compilers umgesetzt in konkrete Steuerimpulse für Gleisnetzelemente nach Routen. Unser Beispielsignal `s3001` etwa muss die Routen `route01`, route02, und route03 freischalten und wieder sperren, wenn sie befahren werden sollen. Konktete Signal-Steuerimpulse für die einzelnen Routen dementsprechend dem TRACS Steuerinterpreter zur Verfügung gestellt.

```
clearances  {
   route01: s3001 straight;
   route02: s3001 straight;
   route03: s3001 right;
   ...
}
```

Analog werden auch Weichenstellungen pro Route ausgewertet.

7.4.2.1.3 Konfliktinformationen Innerhalb der TRACS *TND* sind Informationen darüber enthalten, welche Routen gleichzeitig befahren werden dürfen, und welche Routen bei gleichzeitiger Befahrung kollisionsgefährdet sind. Aus diesen Informationen entsteht eine binäre Tabelle, die es dem TRACS Steuerinterpreter ermöglicht, effizient zu entscheiden, welche angeforderten Routen freigegeben werden dürfen. Es folgt ein Beispiel für solche Konflikinformationen:

```
point-conflicts {
   route01: route02, route03, route06;
   route02: route01, route03, route05;
   route03: route01, route02;
   ...
}
conflicts {
   route01: route04, route05, route06, route08, route10, route12;
```

```
   route02: route04, route05, route06, route07, route08;
   route03: route08, route12;
   ...
}
```

Zusätzlich werden vom TRACS Steuerinterpreter Informationen benötigt, welche Route an welchem Sensor angefordert werden kann. Da dies mehrere Routen pro Anforderungssensor sind, werden gegebenenfalls auch Anforderungs-Warteschlangen benötigt. Auch diese Informationen werden dem TRACS Steuerinterpreter in binärer Form zur Verfügung gestellt. Ein Beispiel verdeutlicht dies:

```
routedefinitions {
   route01: g4101, g4105 - request-at: g4001;
   route02: g4101, g4107 - request-at: g4001;
   route03: g4101, g4102 - request-at: g4001;
   ...
}
```

Für detailliertere Informationen zur *TND - Tram Network Description*, welche die Eingabe für den *tram* Compiler bildet, wird auf Abschnitt 7.1 verwiesen.

7.4.2.2 Ausgabedaten

Die Ausgabedaten des *tram* Compilers bilden die binären Projektierungsdaten, die ein TRACS Steuerinterpreter zum sicheren Steuern eines Gleisnetzes benötigt. Diese sind in drei zentrale Blöcke aufgeteilt, die vom *tram* Compiler separat generiert werden. Diese drei Komponenten der Projektierungsdaten entsprechen der Projeketierung der drei zentralen Komponenten des TRACS Steuerinterpreters. Für genauere Informationen zum TRACS Steuerinterpreter und dessen Architektur wird auf Abschnitt 7.5 verwiesen.

7.4.2.2.1 Projektierung des Route Dispatcher Der Route Dispatcher des TRACS Steuerinterpreters entscheidet zur Laufzeit anhand der vorliegenden Befahrungs- und Anforderungskonstellation des zu steuernden Gleisnetzes und einer entsprechenden Konflikttabelle, welche Routen freigeschalter werden dürfen, und welche Routen momentan nicht befahren werden dürfen. Die dazu notwendige Konflikttabelle, welche beschreibt, welche Routen nicht gleichzeitig befahren werden dürfen, erhält der TRACS Steuerinterpreter aus den Projektierungsdaten. Zusätzlich benötigt man noch noch den Block *Route Definitions* aus der *TND* Beschreibung um die jeweiligen Route Request Sensoren zu bestimmen. Eventuell notwendige Warteschlangen für mehrere Routenanforderungen an einem Sensor werden ebenso aus den Projektierungsdaten ausgelesen. Diese Daten werden aus den oben beschriebenen Konfliktinformationen der vom *tram* Compiler eingelesenen *TND* generiert.

7.4.2.2.2 Projektierung der Route Controller Pro Route existiert in einem laufenden TRACS Steuerinterpreter je ein Route Controller. Diese benötigen zum Steuern ihrer zugeordneten Route alle erforderlichen Steuerimpulse zum Freigeben, Einstellen, und Sperren dieser Route. Hierzu werden binäre Listen von Steuerimpulsen für entsprechende Gleisnetzelemente generiert, die jeweils eine Route Freigeben, ihre Weichen setzen, oder die Route wieder sperren. Diese Steuerimpulse werden pro Route vom *tram* Compiler generiert und in einem effizient auszuwertendem Format dem Steuerinterpreter zur Verfügung gestellt. Als Basis dieser Daten dienen die oben beschriebenen Routeninformationen einer vom *tram* Compiler einzulesenden *TND*.

7.4.2.2.3 Projektierung des Safety Monitor Der Safety Monitor eines TRACS Steuerinterpreter prüft zu jedem Zeitpunkt der Ausführung, ob sich ein gesteuertes Gleisnetz in einem sicheren Zustand befindet. Hierzu werden pro möglicher sicherer Befahrungskonstellation des zu prüfenden Gleisnetzes Sensorrelationen, Weichen- und Signalstellungen, und Fehlermeldungen der Elemente ausgewertet. Dies geschieht indem der Steuerinterpreter eine Liste von sicherern Gleisnetzzuständen auswertet. Befindet sich das Gleisnetz in keinem der gegebenen sicheren Zustände, reagiert der Steuerinterpreter mit seiner Ausnahmebehandlung. Diese Liste von sicheren Zuständen erhält der TRACS Steuerinterpreter aus seiner Projektierung, die der *tram* Compiler zu diesem Zwecke generiert.
Hierzu errechnet der *tram* Compiler zunächst sämtliche sicheren Befahrungskonstellationen. Dies geschieht auf Basis der oben beschriebenen Konfliktinformationen einer eingelesenen *TND*.
Pro sicherer Befahrungskonstellation wird ermittelt, wie sich Sensorenstände des Gleisnetzes zueinander verhalten müssen, und welche Stellung Signale und Weichen haben müssen. Diese Daten entstehen aus den oben beschriebenen Routeninformationen einer *TND*.
Zusätzlich zu Signal- Sensor- und Weichenständen beinhaltet jeder sichere Zustand eine Prüfung aller eingesetzten Gleisnetzelemente auf Fehlermeldungen. Dies wird durch die oben beschriebenen Elementinformationen einer *TND* möglich.
Für genaue Informationen über das Projektierungsdatenformat des TRACS Steuerinterpreters möge der Abschnitt 7.5 zu Rate gezogen werden.

7.4.2.3 Sicherheitsanforderungen

Der *tram* Compiler ist zwar verantwortlich für die konkrete Steuerungslogik eines Gleisnetzes, muss allerdings trotzdem nicht formal validiert werden. Innerhalb des TRACS Verifikationsprozesses existiert ein Modellvergleich, der die vom *tram* Compiler generierte Projektierungssdaten auf Sicherheit prüft. Dieser Modellvergleich trifft vor dem Einsatz von generierten Projektierungsdaten mit dem generischen TRACS Steuerinterpreter die endgültige Entscheidung, ob die für ein konkretes Gleisnetz generierten

Projektierungsdaten im Zusammenspiel mit dem TRACS Steuerinterpreter ein sicheres Steuerungssystem ergeben.
Dennoch stellt der *tram* Compiler den informellen Anspruch an sich, für jedes Gleisnetz sichere Projektierungsdaten zu generieren oder, sollte dies nicht möglich sein, eine entsprechende Fehlermeldung zu generieren.
Nähere Information zur Sicherheitsprüfung von binären Projektierungsdaten finden sich in Abschnitt 7.6.

Helge Löding

7.4.3 Architektur

Der TRACS *tram* Compiler besteht aus modular entwickelten Komponenten, deren Zustammenspiel die Gesamtfunktionalität ausmacht. Hierbei kommen neben der Programmiersprache *C* noch die Compilerbau Standardwerkzeuge *flex* und *bison* zum Einsatz.
Im folgenden wird zunächst der grobe Aufbau des *tram* Compilers mit seinen einzelnen Komponenten beschrieben, bevor auf einzelne Komponenten genauer eingegangen wird.

7.4.3.1 Gesamtaufbau

Generell besteht der *tram* Compiler zunächst aus einem Parser, welcher eine eingegebene *TND* auf syntaktische Korrektheit prüft. Ist eine Eingabedatei korrekt, so wird eine Symboltabelle angelegt, welche alle beschriebenen Gleisnetzelemente sowie deren physikalische Eigenschaften beschreibt und speichert. Zusätzlich wird ein Syntaxbaum angelegt, der die eingelesenen Informationen, die über Hardwareeigenschaften hinausgehen, im Speicher repräsentiert.
Nachdem eine Eingabedatei von der Parser-Komponente des *tram* Compilers erfolgreich verarbeitet wurde, bilden der entstandene Syntaxbaum und die Symboltabelle die Basis für die darauf folgende Generierung von binären Projektierungsdaten.
Eine in *C* geschriebene Bibliothek von Funktionen ist dann dafür verantwortlich, aus dem im Speicher entstandenen Syntaxbaum mit Symboltabelle Projektierungsdaten für den TRACS Steuerinterpreter zu erstellen und diese im Dateisystem abzulegen.
Abbildung 7.29 zeigt den Aufbau des TRACS *tram* Compilers.
Im folgenden wird der Aufbau der einzelnen Komponenten des *tram* Compilers näher betrachtet. An dieser Stelle soll hierbei nur die Struktur des *tram* Compilers erläutert werden. Genaurer Informationen zur Funktionsweise und den benutzten Algorithmen der einzelnen Komponenten finden sich im nächsten Abschnitt 7.4.4.

7.4.3.2 Parser

Der *tram* Compiler benutzt einen automatisch generierten Parser, um eine gegebene *TND* einzulesen. Hierbei werden die Standard Compilerbau Werkzeuge `flex` und `bison` eingesetzt, um einen Parser automatisch generieren zu können.

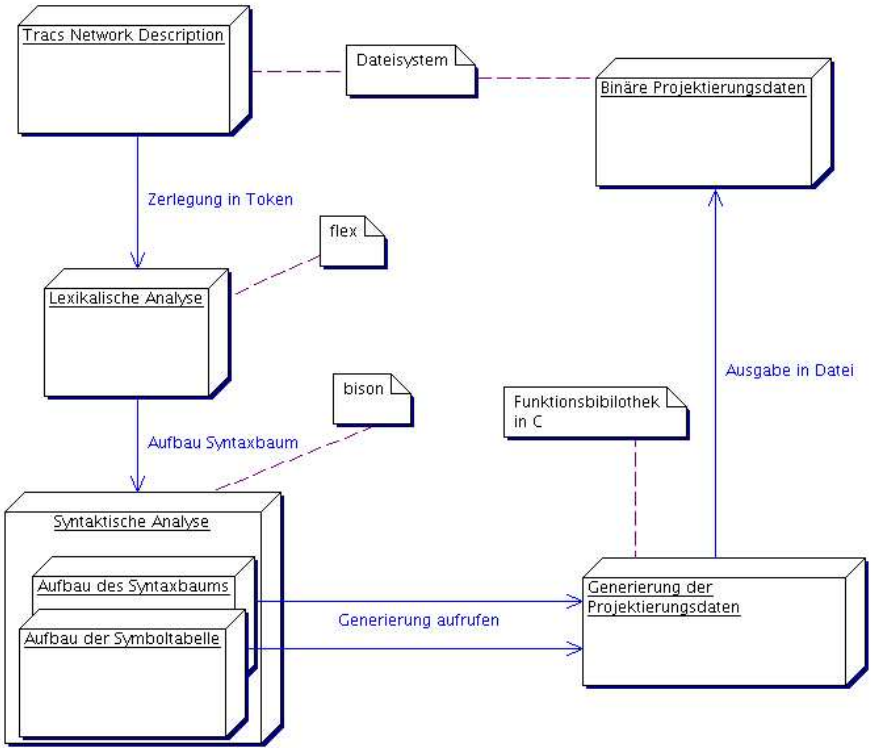


Abbildung 7.29: Übersicht tram Compiler

Mithilfe des Werkzeugs `flex` wird zunächst ein Lexer zur Zerlegung der Eingabedaten in einzelne Token generiert. Hierbei entsteht die Komponente des Parsers, die für die lexikalische Analyse verantwortlich ist, also Schlüsselworte, Elementbezeichner, und Blockmarkierungen unterscheidet, und diese als einzelne Token an die Parserkomponente für die syntaktische Analyse liefert. Hierbei wird mithilfe des Parsergenerators `bison` automatisch ein Parser generiert, welcher gemäß einer gegebenen Grammatik entscheidet, ob die vom Lexer gelieferten Token in einer grammatikalisch korrekten Reihenfolge geliefert werden, die Eingabedatei also korrekt aufgebaut ist.
Der Lexergenerator `flex` erhält als Eingabe die Datei `tram.l`, welche in Form von regulären Ausdrücken definiert, welche Token der zu generierende Lexer unterscheiden soll. Wird ein Token erkannt, so liefert der generierte Lexer neben der eigendlichen Token-Information den Inhalt des Tokens als String an den Parser, da aus diesen Informationen anschliessend ein Syntaxbaum mit Symboltabelle aufgebaut werden soll. Die Tokendefinitionen für den Lexergenerator `flex` entsprechen den regulären Ausdrücken, die in Abschnitt 7.1 die lexikalischen Regeln der TRACS *TND* definieren. Der folgende Ausschnitt aus der `tram.l` zeigt, wie für einen Weichenbezeichner ein regulärer Ausdruck definiert wird, und wie der tatsächliche Bezeichnername zusätzlich zum Tokennamen an den Parser weitergeleitet werden.

```
...
"w"[[:alnum:]]+          { yylval.cp = malloc(strlen(yytext) + 1);
                           memcpy(yylval.cp,
                                  yytext,
                                  strlen(yytext) + 1);
                           return(POINTID); } /* Ausdruck Weichen */
...
```

Die vom Lexer gelieferten Token werden von einem Parser auf syntaktische Korrektheit überprüft, der mit `bison` generiert wird. Dieser Parsergenerator erhält als Eingabe die Datei `tram.y`, in der die zu erkennende Grammatik spezifiziert ist. Zusätzlich zu den einzelnen Regeln der zugrundeliegenden Grammatik finden sich in dieser Datei Auswertungsregeln, die pro Grammatikregel aus den vom Lexer gelieferten Token-Strings durch entsprechende Funktionsaufrufe einen Syntaxbaum und eine Symboltabelle erstellen. Hierbei existiert für jedes nicht-triviale Nicht-Terminal der Grammatik eine Funktion, die für das erkannte Nicht-Terminal seine Entsprechung innerhalb des Syntaxbaums erstellt. Die in `tram.y` definierte Grammatik entspricht der in Abschnitt 7.1 angegebenen Grammatik der TRACS *TND*. Allerdings benötigt `bison` eine Grammatikdefinition in Form einer *Backus-Naur Form*, während die TRACS *TND* in Form einer *Erweiterten Backus-Naur Form* definiert wurde. Eine entsprechende Transformation wurde vorgenommen.
Folgendes Beispiel zeigt eine Grammatikregel, in der einer Weiche eine Richtung zugeordnet wird. Innerhalb der Auswertungsregel wird ein entsprechender Teilbaum des Syntaxbaumes angelegt und zur Weiterverarbeitung an ein höhergelegenes Nicht-Terminal weitergegeben.

```
condition: POINTID direction  { $$ = makeCondClear($1, $2); }
         ;
```

Genauere Informationen zu den Werkzeugen `flex` und `bison` sowie Informationen zu Grammatikdefinitionen und Auswertungsregeln finden sich in [ASU99].

7.4.3.3 Symboltabelle

Die Symboltabelle des *tram* Compilers enthält als Einträge einzelne Gleisnetzelemente in Form von Elementbezeichnern gemäß der eingelesenen TRACS *TND*, sowie deren zugeordnete Hardware-Eigenschaften. Diese Hardware-Eigenschaften gehen aus den `Properties`-Blöcken der TRACS *TND* hervor.
Die *tram* Symboltabelle wird durch die POSIX-Standardfunktionen zum Hashing verwaltet und bietet dementsprechend schnellen Zugriff auf einzelne Einträge der Symboltabelle, wobei der Bezeichnername eines Gleisnetzelements hierbei als Schlüssel für den Zugriff dient.

Der *tram* Syntaxbaum selbst enthält dementsprechend selbst keine Bezeichnernamen, da an jedem Blatt des Baumes anstatt eines Bezeichnernamens ein Verweis auf den entsprechenden Eintrag in der Symboltabelle benutzt wird. Auf diese Weise kann bei der Auswertung des Syntaxbaumes in jedem Kontext nicht nur auf Bezeichnernamen, sondern auch direkt auf zugeordnete Hardware-Eigenschaften zugegriffen werden.
In Abbildung 7.30 wird die Speicherstruktur eines einzelnen Symboltabelleneintrags gezeigt. Neben den primären Attributen eines Gleisnetzelements wie Name, Parser-ID, und Ähnlichem, wird hier deutlich, wie räumliche Koordinaten und weichen- bzw. signalspezifische Eigenschaften gespeichert werden.

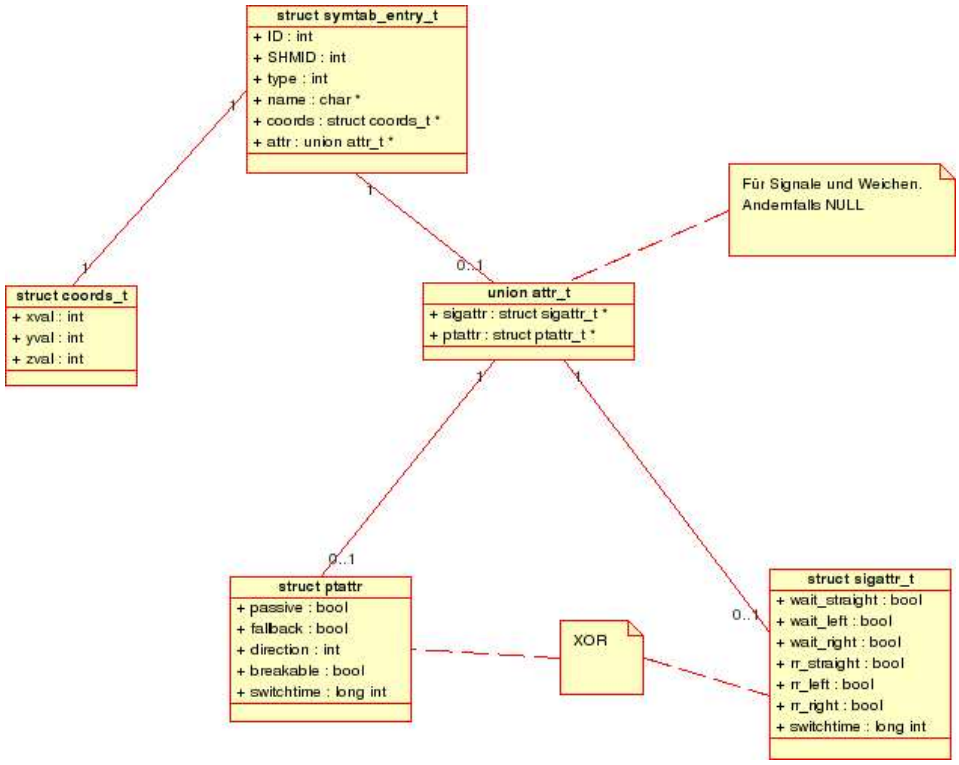


Abbildung 7.30: Übersicht tram Compiler

7.4.3.4 Syntaxbaum

Im Folgenden werden die einzelnen Datenstrukturen beleuchtet, aus denen sich der *tram* Syntaxbaum zusammensetzt.

7.4.3.4.1 struct section_t Der Syntaxbaum des TRACS *tram* Compilers ist auf höchster Ebene aufgebaut als Struktur mit Verweisen auf Teilbäume, die den einzelnen Blöcken der TRACS *TND* entsprechen. Hiervon ausgenommen sind diejenigen Blöcke,

die Hardwareeigenschaften einzelner Gleisnetzelemente beschreiben, da diese Informationen innerhalb der entsprechenden Symboltabelleneinträge gespeichert werden. Abbildung 7.31 zeigt das `struct section_t`, also die Wurzel des *tram* Syntaxbaumes.

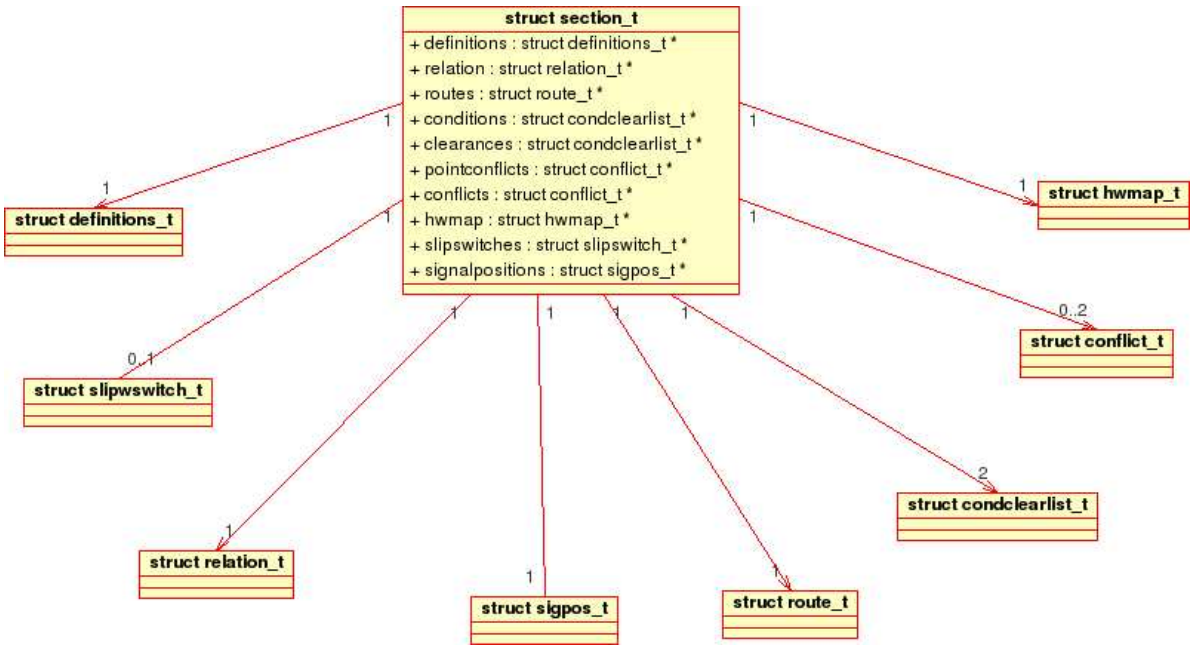


Abbildung 7.31: struct section_t

7.4.3.4.2 struct definitions_t Im `Definitions` Block der TRACS *TND* werden alle Gleisnetzelemente des zu steuernden Gleisnetzes definiert. Zwar finden sich nach dem Parsieren einer eingegebenen *TND* auch alle Gleisnetzelemente in der *tram* Symboltabelle, allerdings ist auch der `Definitions` Block im *tram* Syntaxbaum notwendig, um direkten Zugriff auf Elemente nach Typ zu erhalten. So können durch diesen Teil des Syntaxbaumes etwa alle Links-Geradeaus-Weichen direkt gefunden werden, ohne die Symboltabelle durchsuchen zu müssen. Abbildung 7.32 zeigt den Aufbau dieses Teilbaumes.

7.4.3.4.3 struct slipswitch_t Der `Slipswitch` Block der TRACS *TND* fasst mehrere Weichen zu einer Kreuzungsweiche zusammen. Diese Informationen werden innerhalb der Projektierungsdatengenerierung benötigt, um erkennen zu können, welche Weichen nur gemeinsam geschaltet werden können. Abbildung 7.33 beschreibt die resultierende Speicherstruktur dieses Blockes innerhalb des entstehenden Syntaxbaumes.

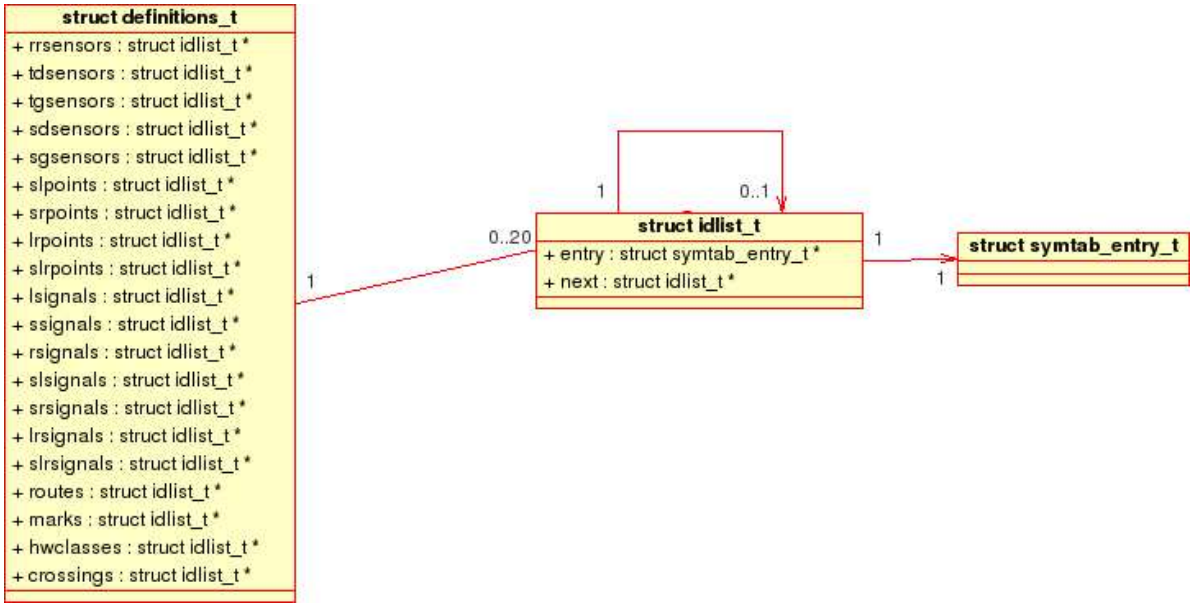


Abbildung 7.32: struct definitions_t

7.4.3.4.4 struct relation_t Der `Relations` Block der TRACS *TND* ordnet Gleisnetzelemente relativ zueinander an. Auf diese Weise können ansonsten uneindeutige Gleisnetzbeschreibungen klarer formuliert werden. Diese Informationen werden primär für Modellprüfungen auf Sicherheit von Projektierungsdaten benötigt. Siehe hierzu auch Abschnitt 7.6. Abbildung 7.34 zeigt den Aufbau dieses Teilbaums des *tram* Syntaxbaumes.

7.4.3.4.5 struct sigpos_t Der `Signalpositions` Block der TRACS *TND* nimmt Zuordnungen zwischen Sensoren und Signalen vor. Dadurch werden insbesondere Eingangssensoren für einzelne Routen deutlich. Diese Informationen werden wie in Abbildung 7.35 gezeigt im *tram* Syntaxbaum abgelegt.

7.4.3.4.6 struct route_t Der `Routedefinitions` Block der TRACS *TND* definiert den Verlauf einzelner Routen durch das zu steuernde Gleisnetz. Hierbei werden die zu überfahrenden Sensoren pro Route aufgeführt. Diese Informationen sind für den TRACS Steuerinterpreter (siehe Abschnitt 7.5) zentral, und finden sich dementsprechend in den vom *tram* Compiler generierten Projektierungsdaten wieder. Abbildung 7.36 zeigt, wie diese Informationen im Speicher dargestellt werden.

7.4.3.4.7 struct condclearlist_t Die *TND* Blöcke `Conditions` und `Clearances` beinhalten Informationen darüber, welche Stellungen Weichen und Signale haben müs-

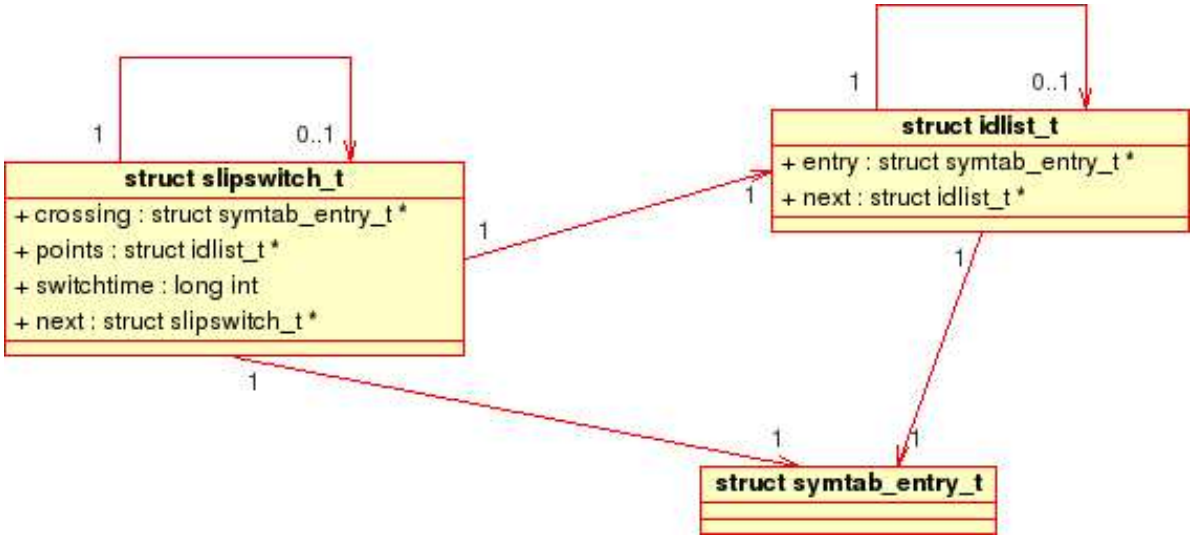


Abbildung 7.33: struct slipswitch_t

sen, um einzelne Routen sicher befahren zu können. Diese Informationen bilden den Grundstock der vom *tram* Compiler zu generierenden Projektierungsdaten, da sie pro Route direkt in Treiberbefehle der entsprechenden Gleisnetzelemente umgesetzt werden. In Abbildung 7.37 wird dargestellt, wie Informationen über Signal- und Weichenstellungen pro Route im *tram* Syntaxbaum abgelegt werden.

7.4.3.4.8 struct conflict_t Die Blöcke `Conflicts` und `Pointconflicts` der TRACS *TND* beschreiben, welche Routen nie gleichzeitig befahren werden dürfen, da auf diesen Routen Kollisionen nicht auszuschliessen wären. Aus diesen Informationen entsteht innerhalb der Projektierungsdatengenerierung der sogenannte *Route Conflict Table*, der vom TRACS Steuerinterpreter genutzt wird, um Routen sicher freigeben zu können. Nähere Informationen hierzu finden sich in Abschnitt 7.5. Abbildung 7.38 zeigt, wie Konflikte zwischen Routen innerhalb des *tram* Syntaxbaumes repräsentiert werden, bevor sie in Projektierungsdaten umgesetzt werden können.

7.4.3.4.9 struct hwmap_t Der Block `Hardwaremap` der TRACS *TND* beschreibt, welche Hardwareelemente von welchen Treibern bedient werden. Hierzu werden sogenannte Hardwareklassen beschrieben, die alle Elemente, die einen Treibertyp nutzen können, in eine Hardwareklasse gliedern. Diese Informationen ordnen den Gleisnetzelementen die zu nutzenden Treiber zu. Abbildung 7.39 zeigt die Struktur dieser Daten innerhalb des Syntaxbaumes.

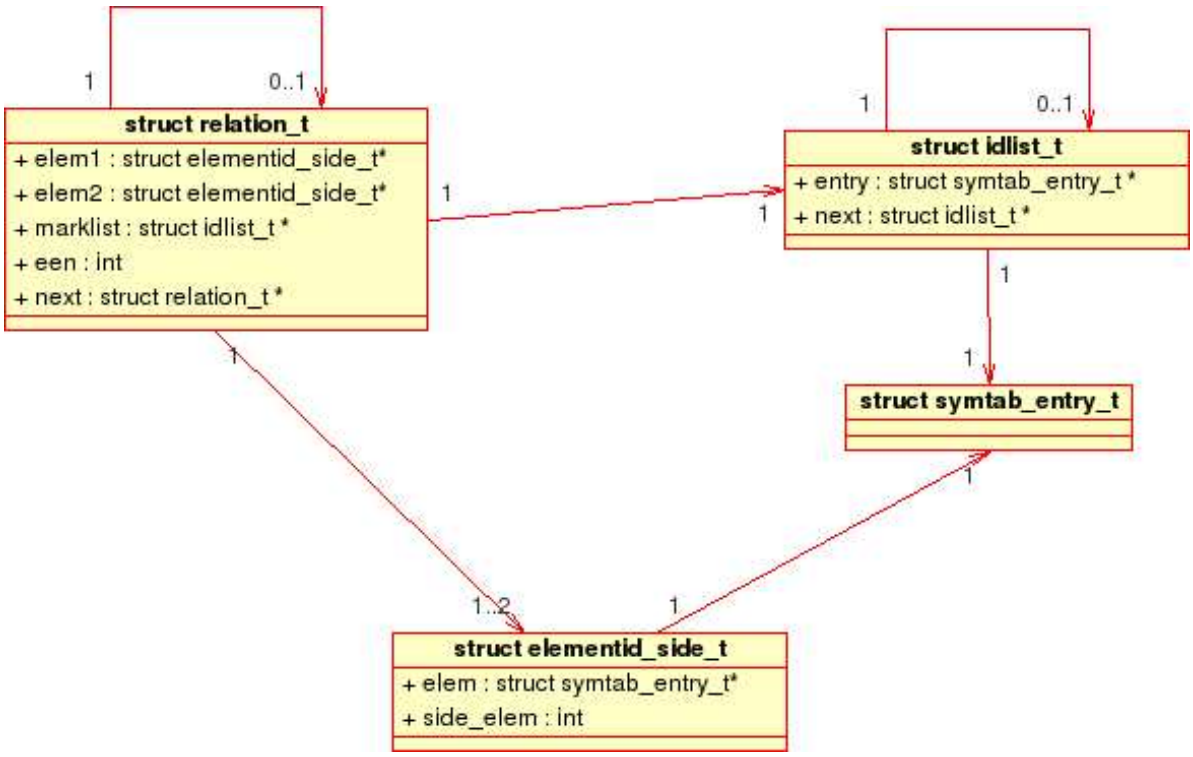


Abbildung 7.34: struct relation_t

7.4.3.5 Projektierungsdaten

Die zu generierenden binären Projektierungsdaten werden vom *tram* Compiler zunächst innerhalb des Speichers erzeugt. Hierbei wird die selbe Speicherstruktur erzeigt, wie sie vom TRACS Steuerinterpreter eingelesen und genutzt wird. Allerdings werden Inline-Pointer und Offsets erst zum Zeitpunkt des Schreibens der Projektierungsdaten in das Dateisystem errechnet.
Genauere Informationen zur Speicherstruktur der zu generierenden Projektierungsdaten eines TRACS Steuerinterpreters finden sich daher im entsprechenden Abschnitt 7.5.

Helge Löding

7.4.4 Algorithmen

In diesem Abschnitt soll nun das Vorgehen des *tram* Compilers bei der Erzeugung der Projektierungsdaten gezeigt werden. Dabei wird jeder Verarbietungsschritt des *tram* Compilers vorgestellt und den einzelnen Komponenten des *tram* Compilers zugeordnet. Als erstes wird der Parser vorgestellt, der eine *TND* Textdatei einliest. Gofelgt wird dieser Arbeitsschritt vom Aufbau des Syntaxbaums. Anschließend soll noch genauer auf die Generierung der Projektierungsdaten eingegangen werden.

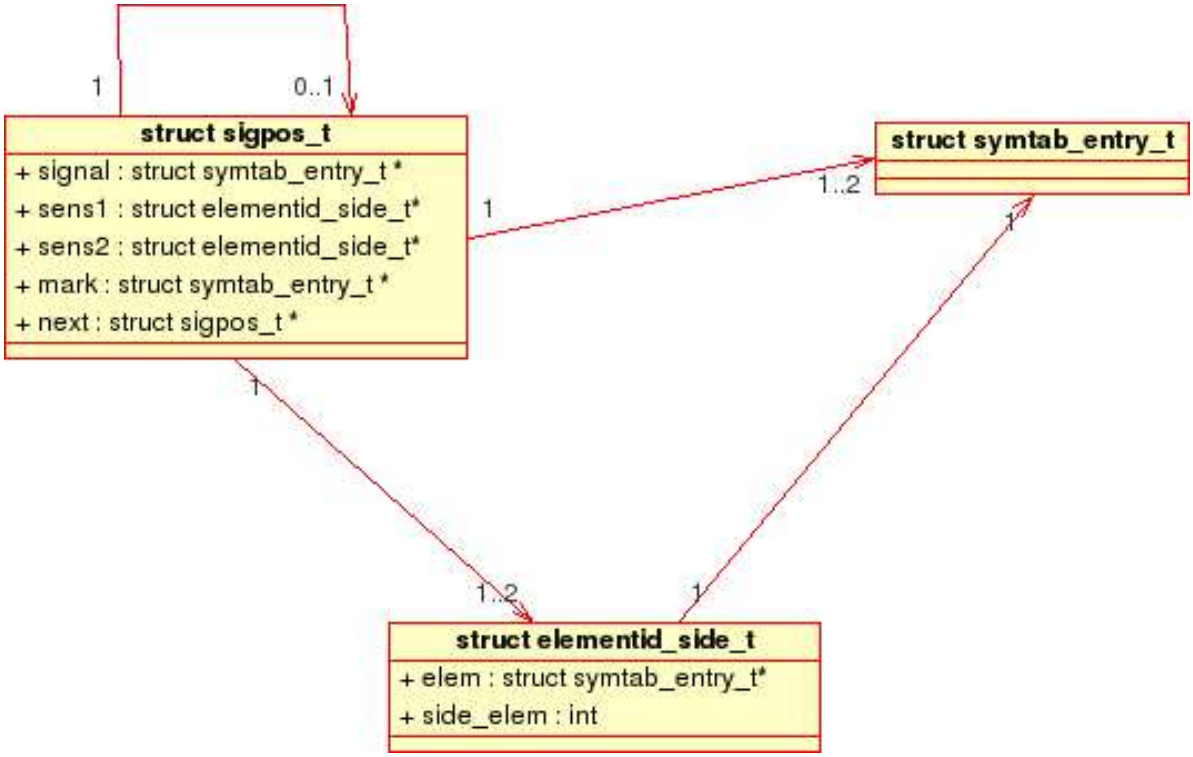


Abbildung 7.35: struct sigpos_t

7.4.4.1 Parser

Hier soll nun der Parser mit seiner Funktionsweise erläutert werden. Der Parser erhält vom Scanner eine Folge von Symbolen, die dieser aus der *TND* Beschreibung eines Gleisnetzes erzeugt. Diese Symbolfolge wird von ihm geprüft, ob sie auch der Grammatik der für die *TND* erzeugten *EBNF* entspricht. Bei vorkommenden Fehlern wird der Parser eine Fehlermeldung generieren und diese mit der Zeilenangabe des Fehlers ausgeben. Dabei geht er nach der *panic recovery* Methode vor, d.h. bei einem gefunden Fehler versucht der Parser an einer geeigneten Stelle das Parsen wieder aufzunehmen damit er möglichst viele Fehler in einem Durchgang finden kann. Der Parser des *tram* Compilers ist ein *BOTTOM -UP* Parser, d.h. bei der Ereugung des Syntaxbaums wird an den Blättern angefangen und der Parser arbeitet von dort bis oben zur Wurzel durch. Der Parser versucht dabei immer die rechte Seite einer Produktionsregel mit der linken Seite zu reduzieren, bzw. durch das Symbol auf der linken Seite dieser Produktion zu ersetzen. Genauer gesagt handelt es sich bei dem Parser um einen LALR1-Parser. Diese Bezeichnung setzt sich folgendermaßen zusammen. Das LA steht für *lookahead*, also daß der Parser voprausschauend arbeitet. Das nächste L steht dafür, daß der Parser die Eingabe von *links nach rechts* abarbeitet. Das R steht dafür, daß eine Rechtsableitung

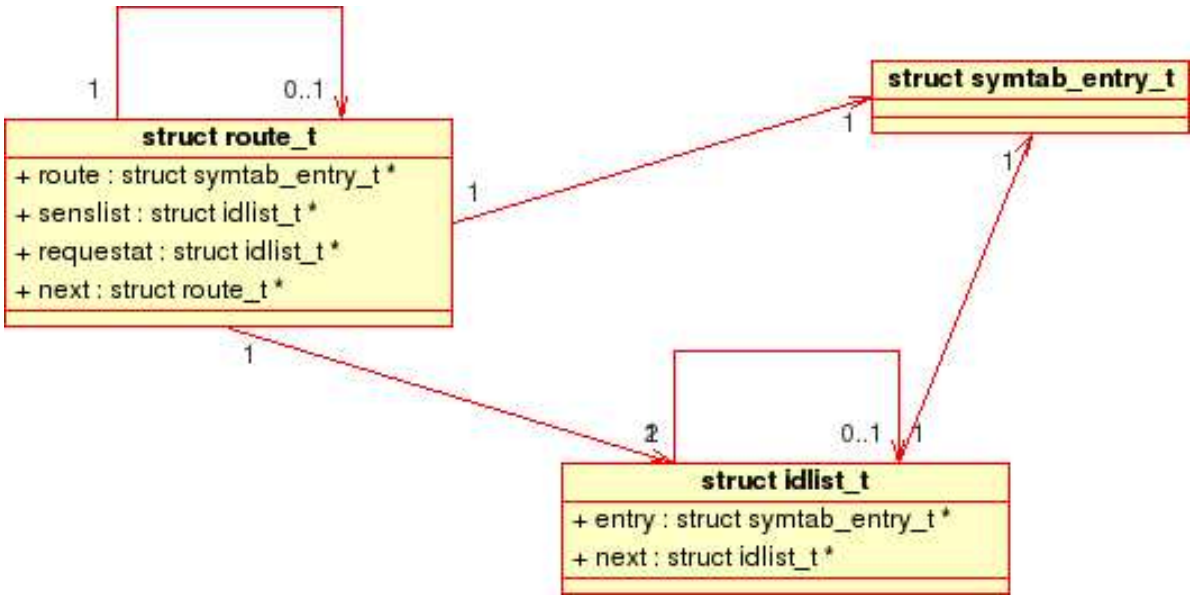


Abbildung 7.36: struct route_t

in umgekehrter Reihenfolge gebildet wird und die 1 steht dafür, daß ein Eingabesymbol im voraus betrachtet wird. Zur genaueren Arbeitsweise eines solchen Parsers finden sich in [ASU99] genauere Informationen.

7.4.4.2 Syntaxbaum

In diesem Abschnitt soll die Funktionsweise des Parsers genauer beschrieben werden. Hierbei soll auf die Grammatikregeln und auf die vorhandenen Auswertungsregeln eingegangen werden, die letztendlich die Symboltabelle und den Syntaxbaum erstellen, die zur weiteren Erzeugung der Projektierungsdaten unabdingbar sind. Die Grammatik des *tram* Compilers entspricht ziemlich der aus der *TND*, hierzu wurde nur eine Transformation von der *EBNF* Form zu einer *BNF* Form durchgeführt, da *BISON* nur diese versteht. Da es bei dem Parser um einen *BOTTOM-UP* Parser handelt wird der Syntaxbaum von den Blättern aus zur Wurzel aufgebaut. Dies geschieht durch miteinander verknüpfte structs, die den Knoten des Baums entsprechen. Um hier einen besseren Überblick zu gewährleisten sei hier dennoch die Wurzel als erstes dargestellt.

```
tnd: defblock coordblock sigpropblock ptpropblockopt
   slipswitchblockopt relblock sigposblock routeblock
   condblock clearblock ptconfblockopt confblockopt hwblock{

  syntree = makeSyntree();
```

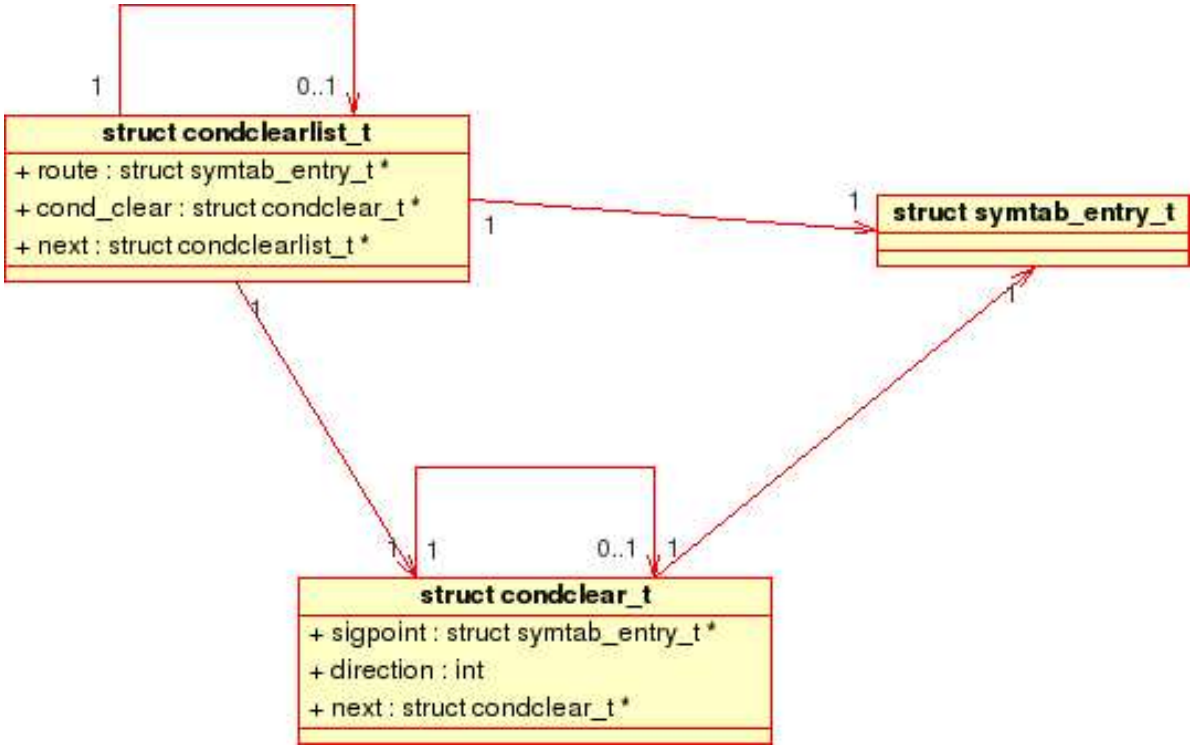


Abbildung 7.37: struct condclearlist_t

```
}
    ;
```

Dies ist der letzte Verarbeitungsschritt des Parsers der alle vorhanden Blöcke schon reduziert hat. Diese Blöcke werden nun zu der Wurzel des Syntaxbaums reduziert. Die einzelnen Blöcke entsprechen denen der *TND*, deren Name hier nur der Übersichtlichkeit gekürzt wurden. Da einige Blöcke optional sind tragen sie als Konvention opt am Ende ihres Namens. Mit dem Aufruf der folgenden Funktion werden die einzelnen Blöcke an die richtige Stelle des Syntaxbaums hinzugefügt.

```
struct section_t *makeSyntree(struct definitions_t *definitions,
      struct slipswitch_t *slipswitches,
      struct relation_t *relations,
      struct sigpos_t *signalpositions,
      struct route_t *routes,
      struct condclearlist_t *conditions,
      struct condclearlist_t *clearances,
```

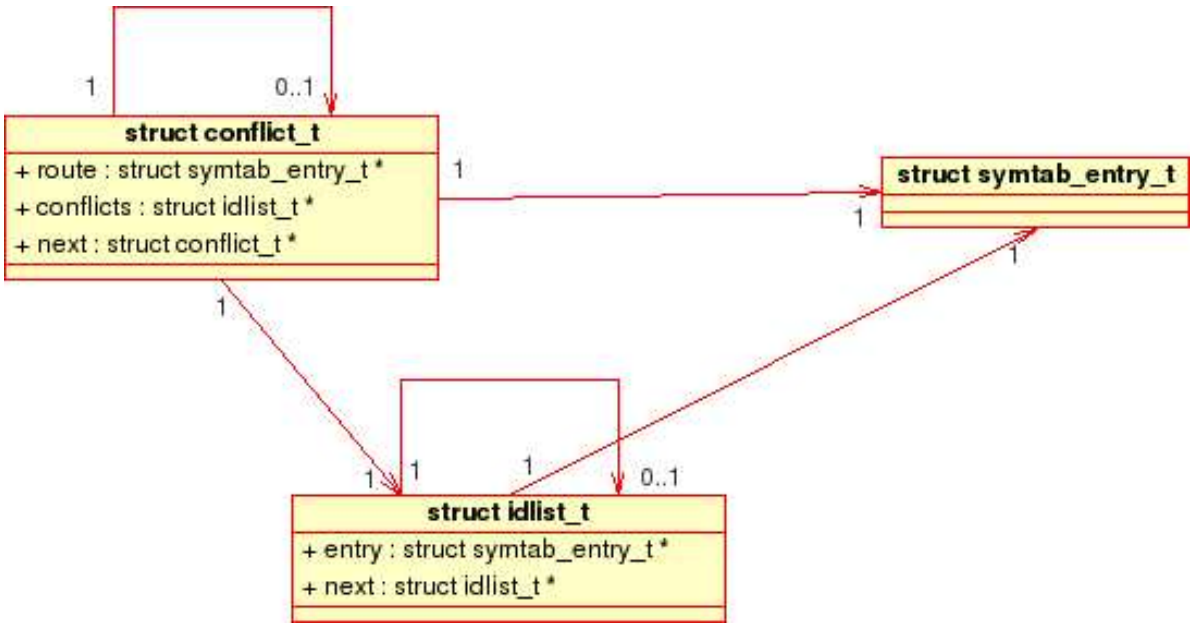


Abbildung 7.38: struct conflict_t

```
        struct conflict_t *conflicts,
        struct conflict_t *pointconflicts,
        struct hwmap_t *hwmap);
```

Als Regel die einen Block optional macht sei hier nur die für die point conflicts aufgeführt, da die anderen optionalen Blöcke exakt auf die selbe Weise optional gehalten werden.

```
ptconfblockopt: ptconfblock
              | /* empty */
              ;
```

Diese Regel sagt nur aus, daß ein ptconfblock vorkommen kann und dann zu ptconfblockopt reduziert wird oder ansonsten ptconfblockopt leer ist.
Als nächstes folgen die Auswertungen der einzelnen Blöcke, angefangen bei den definitions.

```
defblock: DEFINITIONS '{' typedeflist '}'
        ;
```

Wenn das Schlüsselwort *Definitions* gefolgt von einer in geschweiften Klammern stehenden *typedeflist* erkannt wird wird diese zum *defblock* reduziert. Dazu muß zunächst aber erst einmal eine *typedeflist* erkannt werden. Dies geschieht dann durch folgende Regel.

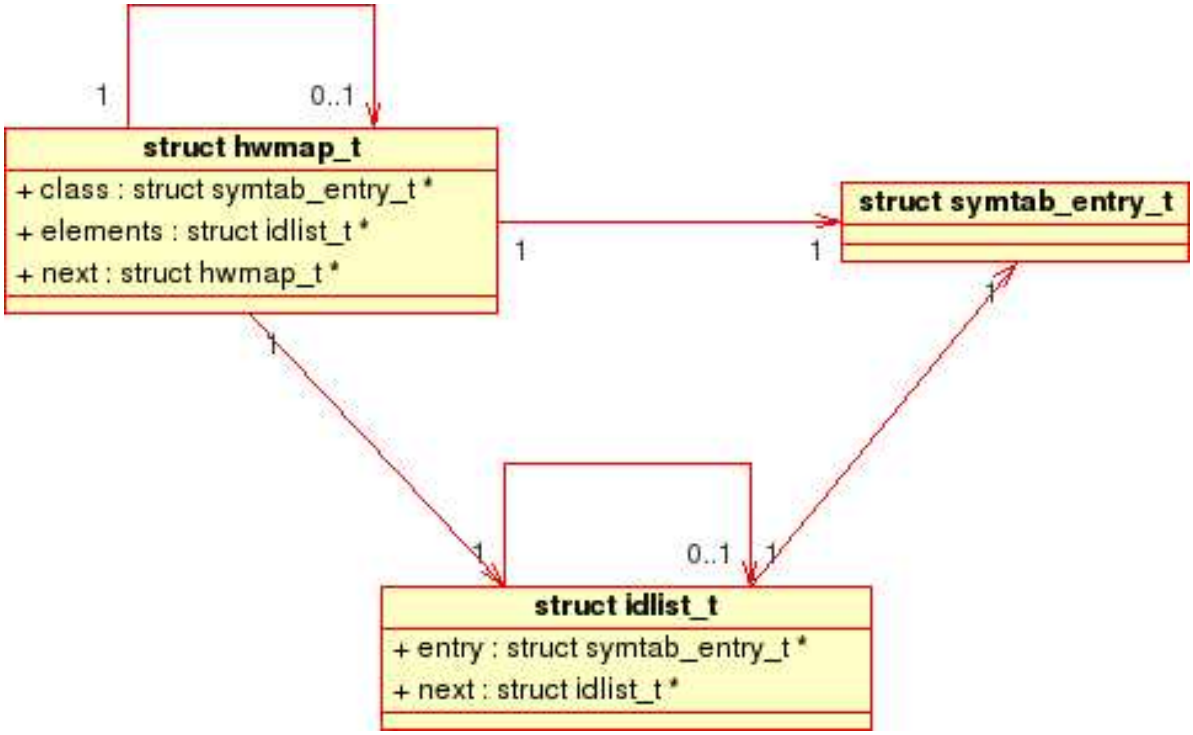


Abbildung 7.39: struct hwmap_t

```
typedeflist: typedef { fillDefinitions(); }
           | typedef typedeflist{ fillDefinitions(); }
   ;
```

Eine *typedeflist* besteht aus einer *typedef* oder auch aus mehreren. Dies wird durch die Rekursion in der zweiten Regel ermöglicht. Durch den Aufruf von

```
struct definitions_t *fillDefinitions(struct definitions_t *defs,
                      struct idlist_t *list);
```

werden die Daten in den Syntaxbaum eingetragen. Hierfür ist es für den Parser wichtig vorher bereits *typedef* erkannt zu haben.

```
typedef: senstype ':' sensdef ';'  { fillType(); }
       | pointtype ':' pointdef ';'  { fillType();  }
       | sigtype ':' sigdef ';' { fillType(); }
       | ROUTES ':' routedef ';'
       | MARKS ':' markdef ';'
       | HWCLASSES ':' hwdef ';'
       | CROSSINGS ':' crossdef ';'
       ;
```

Bei *typedef* handelt es sich um die Beschreibung aller Elemente des Gleisnetzes. Wobei nur eine näher beschrieben werden soll, da die Regeln für die anderen Elemente ähnlich aufgebaut sind. Ein *typedef* kann zum Beispiel ein *senstype* gefolgt von einem Doppelpunkt und einem *sensdef* mit einem abschließenden Semikolon sein. Unter *senstype* versteht man die verschieden Sensortypen, die vorkommen können. Dies ist in der folgenden Reduktionsregel verankert.

```
senstype: RRSENSORS
        | TDSENSORS
| TGSENSORS
| SDSENSORS
| SGSENSORS
;
```

Das heißt, daß der *senstype* immer aus einem Schlüsselwort besteht, der den Sensortypen repräsentiert. Die Reduktion zu *sensdef* ist ein wenig komplizierter.

```
sensdef: SENSID { makeList(); }
       | SENSID ',' sensdef { makeList(); }
       ;
```

Hierbei kann wieder auch eine Liste von mehrerer Bezeichner für den jeweiligen Sensortypen erstellt werden. Die Funktion

```
struct idlist_t *makeList(char*,
  struct idlist_t*,
  int);
```

Für Weichen und Signale funktioniert dies analog. Bei den Routen, Markierungen, Hardwareklassen und Kreuzungen ist dies ähnlich, nur ist bei denen eine Reduktion weniger nötig, da es hier nur ein Schlüsselwort gibt und nicht noch nach dem genauen Typen geschaut werden muß. Das wären auch alle Regeln für den Definitionsblock.
Der zweite Block ist der *coordblock*. Eine Reduktion auf diesen wird durch folgende Regeln erreicht.

```
coordblock: COORDINATES '{' coorddeflist '}'
          ;
```

Der *coordblock* besteht aud dem hierfür vorgesehenen Schlüsselwort gefolgt von einer in geschweiften Klammern stehenden *coorddeflist*, die es wieder nur ermöglicht eine Listenstruktur aus mehreren *coorddef* zu machen. Zu einem *coorddef* kann wiederum nach folgender Regel reduziert werden.

```
coorddef: elementid ':' '(' coord ',' coord ',' coord ')' ';'
          { fillCoords(); }
        ;
```

Sie besteht aus einer elementid und den zugehörigen Koordinaten. Durch den Aufruf von

```
void fillCoords(char *elem,
int x, int y, int z);
```

wird das entsprechende Element aus der Symboltabelle herausgesucht und die Koordinaten hierfür in diese eingetragen. Da es verschiedene Elementtypen geben kann muß folgendermaßen zu *elementid* reduziert werden.

```
elementid: SENSID
         | POINTID
         | SIGID
 | MARKID
         | CROSSID
         ;
```

Hierbei besteht eine *elementid* aus dem zugehörigen Bezeichner für ein bestimmtes Element. Verbleibt nur noch die Reduktion der eigentlichen Koordinaten auf *coord.*Dieses geschieht auf diese Weise.

```
coord: POSINT
     | NEGINT
     ;
```

Wobei POSINT und NEGINT die entsprechenden Werte für die Koordinaten beinhalten. Der Signal-Properties Block wird folgendermaßen reduziert.

```
sigpropblock: SIGNAL_PROPERTIES '{' sigpropdeflist '}'
            ;
```

Er besteht also aus dem Schlüsselwort für die Signal-Properties gefolgt von einer in geschweiften Klammern stehenden *sigpropdeflist.* Diese ist wiederum folgendermaßen aufgebaut.

```
sigpropdeflist: sigpropdef
              | sigpropdef sigpropdeflist
      ;
```

Hierbei wird wieder eine Listenstruktur ermöglicht, da es eine oder mehrere *sigpropdef* geben kann. Eine *sigpropdef* kann reduziert werden, wenn folgende Regel anzuwenden ist.

```
sigpropdef: SIGID ':' waitstraightopt waitleftopt waitrightopt
                      rrstraightopt rrleftopt rrrightopt
                      SWITCHTIME POSINT ';'
{ insertSignalProperties(); }
          ;
```

Sie besteht also aus einem Bezeichner *SIGID* und den zugehörigen Eigenschaften. Durch den Aufruf von

```
void insertSignalProperties(char *name, bool waits, bool waitl,
                            bool waitr, bool rrs, bool rrl,
                            bool rrr, long int switchtime);
```

werden die Eigenschaften für das entsprechende Signal in die Symboltabelle eingetragen. Hierbei sind einige Eigenschaften optional. Dies wird wie folgt ermöglicht.

```
waitstraightopt: WAIT_STRAIGHT
               | /* empty */
               ;
```

Gibt es das entsprechende Schlüsselwort für die Eigenschaft wird der Wert hierfür auf TRUE gesetzt, ansonsten auf FALSE. Dies geschieht entsprechend bei allen optionalen Eigenschaften.
Der nächste eventuell zu reduzierende Block ist der für die Point Properties. Dessen höchste Regel schaut nach dem Schlüsselwort und dessen anhängende Liste von *ptpropdef*.

```
ptpropblock: POINT_PROPERTIES '{' ptpropdeflist '}'
           | POINT_PROPERTIES '{' error '}'
           ;

ptpropdeflist: ptpropdef
             | ptpropdef ptpropdeflist
     ;
```

Eine *ptpropdef* besteht aus dem Bezeichner für die Weiche und denn Eigenschaften.

```
ptpropdef: POINTID ':' passiveopt fallbackopt breakableopt
           SWITCHTIME POSINT ';'
{ insertPointProperties(); }
         ;
```

Wobei wieder der Postfix opt für die optionalen Eigenschaften steht, die wie bereits vorher beschrieben optional gehalten werden. Durch den Aufruf von

```
void insertPointProperties(char *name, bool psv,
   int dir, bool brk, long int sw);
```

wird das entsprechende Element aus der Symboltabelle ausgesucht und die zugehörigen Eigenschaften dort eingetragen. Als nächstes soll die Auswertung des Slipswitch Blockes erläutert werden. Zum *slipswitchblock* kann auf folgende Art und Weise reduziert werden.

```
slipswitchblock: SLIP_SWITCHES '{' slipswitchdeflist '}'
               ;
```

Er besteht aus dem zugehörigen Schlüsselwort und einer *slipswitchdeflist*, die dem Schlüsselwort in geschweiften Klammern stehend folgt. Diese *slipswitchdeflist* kann aus einer oder mehreren *slipswitchdefs* bestehen.

```
slipswitchdeflist: slipswitchdef { makeSlipswitchList(); }
                 | slipswitchdef slipswitchdeflist
{ makeSlipswitchList(); }
 ;
```

Mit dieser Regel wird so eine Listenstruktur ermöglicht. Durch den Aufruf von

```
struct slipswitch_t *makeSlipswitchList(struct slipswitch_t *newss,
struct slipswitch_t *oldlist);
```

wird das neue Element zu der bereits vorhandenen Liste von Slipswitches vorne angehängt. Dazu muß aber vorher zu *slipswitchdef* nach folgender Regel reduziert worden sein.

```
slipswitchdef: CROSSID ':' POINTID ',' POINTID pair2opt ':'
               SWITCHTIME POSINT ';'
{ makeSlipswitch(); }
             ;
```

Hierbei ist *CROSSID* der Bezwichner der Kreuzung, *POINTID* sind die Bezeichner für die Weichen, *pair2opt* kann ein optionales zweites Weichenpaar sein, *SWITCHTIME* ist ein Schlüsselwort und *POSINT* ist der positive Wert den diese annehmen kann. Ein zweites Weichenpaar wird durch folgende Regel auf *pair2opt* reduziert.

```
pair2opt: ',' POINTID ',' POINTID { makeList(); }
        ;
```

Die Funktion

```
struct slipswitch_t *makeSlipswitch(char *cross,
    struct idlist_t *pointlist,
    long int switchtime);
```

erstellt aus den übergebenden Werten einen einzelnen Slipswitch-Eintrag.
Als nächstes folgt die Auswertung des Relation Blocks. Die Reduzierung auf den *relblock* erfolgt wie auch bei den anderen Blöcken durch das Erkennen des zugehörigen Schlüsselworts und der nachfolgenden Liste.

```
relblock: RELATIONS '{' reldeflist '}'
        ;
```

Durch folgende Regel ist es möglich mehrere *reldefs* zu haben.

```
reldeflist: reldef { makeRelationList(); }
          | reldef reldeflist { makeRelationList(); }
  ;
```

Der Aufruf der Funktion

```
struct relation_t *makeRelationList(struct relation_t *newrel,
struct relation_t *oldlist);
```

fügt die einzelnen Relation-Einträge zu einer Liste zusammen. Zu einer einzelnen *reldef* kann nun folgendermaßen reduziert werden.

```
reldef: sensid_side ':' marklistoptkomma elementid_side ';'
{ makeRelation(); }
      | crossid_side ':' marklistoptkomma elementid_side ';'
{ makeRelation(); }
      | pointid_side ':' marklistoptkomma elementid_side ';'
{ makeRelation(); }
      | sensid_side ':' marklistoptkomma een ';'
{ makeRelation(); }
      ;
```

Die Funktion

```
struct relation_t *makeRelation(struct elementid_side_t *elem1,
struct elementid_side_t *elem2,
struct idlist_t *marks,
int een);
```

erhält als Eingaben die beiden Elemente mitsamt ihren Seitenangaben, die Idlist mit den Marks und einen Wert für einen eventuellen Ausgang in dem Gleisnetz und erstellt hieraus einen neuen, einzelnen Relations-Eintrag. Bei dieser Reduktion sind die ersten Nichtterminale, wie sensidside, crossidside und pointidside die jeweiligen Bezeichner der Elemente und seien hier am Beispiel für die crossidside gezeigt.

```
crossid_side: CROSSID cross_side { makeElementidSide(); }
            ;

cross_side: SIDEA
          | SIDEB
          | CROSS_SIDE_C
          | CROSS_SIDE_D
          ;
```

Durch den Aufruf der Funktion

```
struct elementid_side_t *makeElementidSide(char *elem,
   int side)
```

wird ein neuer Elementid-Side-Eintrag erstellt. Für die anderen Side-Einträge funktioniert dies ähnlich. Das Nichtterminal *elementidside* besteht nur aus den anderen Nichtterminalen.

```
elementid_side: sensid_side
              | crossid_side
              | pointid_side
              ;
```

Das Nichtterminal *marklistoptkomma* dient nur dazu das Terminal *marklistkomma* optional zu machen. Die *marklistkomma* kann aus einem oder mehreren Bezeichnern für die Markierungs-Ids bestehen.

```
marklistkomma: MARKID ',' { makeList(); }
             | MARKID ',' marklistkomma { makeList(); }
     ;
```

Die Funktion

```
struct idlist_t *makeList(char *name,
struct idlist_t *oldlist,
int type);
```

erhält als Eingabe den Namen des Elements, die vorhandene Idlist und den Typs des Eintrags. Wenn möglich sucht sie den entsprechenden Eintrag aus der Symboltabelle heraus. Ist dieser nicht vorhanden wird er in die Symboltabelle eingetragen. Anschließend wird der Eintrag der Idlist hinzugefügt. Das Nichtterminal *een* steht für mögliche Gleisnetzbegrenzer anstelle einer zweiten Seitenangabe.

```
een: ENTRANCE
   | EXIT
   | NONE
   ;
```

Als nächstes soll gezeigt werden, wie der Signal-Positions Block ausgewertet wird.

```
sigposblock: SIGPOSITIONS '{' sigposdeflist '}'
           ;
```

Wie bei den anderen Blöcken wird auch hier das entsprechende Schlüsselwort erkannt, das von einer *sigposdeflist* in geschweiften Klammern gefolgt wird. Diese *sigposdeflist* kann aus einer oder mehreren *sigposdefs* bestehen, was wie folgt bewerkstelligt wurde.

```
sigposdeflist: sigposdef { makeSigposList(); }
             | sigposdef sigposdeflist { makeSigposList(); }
     ;
```

Durch den Aufruf von *makeSigposList* werden die Signal-Psitions zu einer Liste zusammengefügt. Dabei bekommt diese Funktion jeweils die Adresse des neuen Eintrags und die Adresse der bereits vorhandenen Liste übergeben. Zu der *sigposdef* kann dann folgendermaßen reduziert werden.

```
sigposdef: SIGID ':' sensid_side '-' MARKID '-' sensid_side ';'
{ makeSigpos(); }
         | SIGID ':' sensid_side ';'
{ makeSigpos(); }
         ;
```

Als erstes kommt die zugehörige Id des Signals mit seiner *sensidside* und gegebenenfalls einer *MARKID* und einer folgenden *sensidside*. Fie Funktion *makeSigpos* erstellt einen einzelnen Signal-Position-Eintrag. Dabei werden hier der Bezeichner des Signals, Die Adresse des ersten Sensors mitsamt Seitenangabe, die Adresse des zweiten Sensors mitsamt Seitenangabe und der Bezeichner für die jeweilige Markierung übergeben. Sind letzere Beiden nicht vorhanden, wird NULL übergeben.
Die letzte Reduktion zum *routeblock* erfolgt gemäß den bereits gezeigten Blöcken.

```
routeblock: ROUTEDEFINITIONS '{' rtdeflist '}'
          ;
```

Es wird auch wie bereits vorher eine Listenstruktur ermöglicht.

```
rtdeflist: rtdef { makeRouteList(); }
         | rtdef rtdeflist { makeRouteList(); }
 ;
```

Die einzige Änderung zu den meisten vorhergehenden Blöcken ist die Reduktion auf *rtdef*, die folgendermaßen aussieht.

```
rtdef: ROUTEID ':' SENSID ',' SENSID kommasenslistopt '-'
       REQUESTAT ':' SENSID kommasenslistopt ';'
{ $$ = makeRoute(); }
```

Als erstes steht der Bezeichner der Route gefolgt von den Bezeichnern zweier Sensoren und einer *kommasenslist*, die eine Liste von weiteren Sensoren ermöglicht, einem Schlüsselwort für einen Requestsensoren und mindestens einem Bezeichner für den entsprechenden Sensoren. Die Funktion *makeRoute* erstellt dann einen einen Eintrag in der Routedefinitions-Liste, wobei sie den Bezeichner für die Route, die Adresse der Idlist für die Sensoren und die Adresse der Idlist der Requestsensoren übergeben bekommt. Wie der Postfix bereits andeutet ist die *kommasenslistopt* optional, daß durch die bereits bekannte Regel realisiert ist.

```
kommasenslistopt: kommasenslist
                | /* empty */
        ;
```

Die *kommasenslist* ist wie die meisten anderen Listenstrukturen aufgebaut.

```
kommasenslist: ',' SENSID { makeList(); }
             | ',' SENSID kommasenslist { makeList(); }
     ;
```

Als nächstes soll gezeigt werden wie der Conditions Block ausgewertet wird. Das geschieht wie sonst auch immer durch ein Schlüsselwort gefolgt von einer Liste.

```
condblock: CONDITIONS '{' conddeflist '}'
         ;
```

Diese Liste wird auch wie üblich ausgewertet.

```
conddeflist: conddef { makeCondClearListNode(); }
           | conddef conddeflist { makeCondClearListNode(); }
   ;
```

Durch die Funktion *makeCondClearListNode* werden Condition-Clearances Listen-Einträge zu einer Liste zusammengefügt. Diese Funktion erhält als Eingaben den einen Condition-Clearances-List-Eintrag und die bereits vorhandene Condition-Clearances-Liste und verknüpft diese dann. Zu *conddef* wird dann folgendermaßen ausgewertet.

```
conddef: ROUTEID ':' condition kommaconditionlistopt ';'
{ makeCondClearList(); }
       ;
```

Diese besteht also aus dem Bezeichner für die Route gefolgt von mindestens einer Condition. Mehrere Conditions werden genauso, wie im zuvorkommenden Block ermöglicht. Bleibt nur noch zu nennen, wie man zu *condition* reduziert.

```
condition: POINTID direction  { makeCondClear(); }
         ;
```

Diese besteht aus einem Bezeichner für eine Weiche und deren entsprechnde Richtung für die Route. Die Richtung ist hierbei aber ein Nichtterminal zu dem auch erst reduziert werden muß. Dies geschieht durch folgende Regel.

```
direction: LEFT
         | RIGHT
 | STRAIGHT
 ;
```

Als nächster Block folgt der Clearances Block. Dieser wird wie immer auf folgende Weise ausgewertet. Dieser wird im Prinzip genause wie der Conditions Block ausgewertet, nur dass hier statt der Bezeichner für die notwendigen Weichen, die Bezeichner für die entsprechenden Signale ausgewertet werden müssen.
Der Point-Conflicts Block ist der nächste auszuwertende Block. Dieser ist den vorherigen auch wieder sehr ähnlich und soll deswegen hier einmal ohne speziellere Erläuterungen gezeigt werden.

```
ptconfblock: POINT_CONFLICTS '{' confdeflist '}'
       ;

confdeflist: confdef { makeConflictList(); }
           | confdef confdeflist { makeConflictList(); }
   ;

confdef: ROUTEID ':' ROUTEID kommaroutelistopt ';'
{ makeConflict(); }
       ;

kommaroutelistopt: kommaroutelist
                 | /* empty */
         ;

kommaroutelist: ',' ROUTEID { makeList(); }
              | ',' ROUTEID kommaroutelist { makeList(); }
      ;
```

Die einzige in ähnlicher Form bisher noch nicht aufgetretene Auswertungsregel ist die für *confdef*. Diese braucht den Bezeichner für die entsprechnende Route, die von einem oder mehreren Bezeichnern der mit ihr in Konflikt stehenden Routen gefolgt wird.
Der Conflicts Block sieht noch einfacher aus, da er dem Point-Conflicts Block nahezu gleicht.

```
confblock: CONFLICTS '{' confdeflist '}'
         ;
```

Er hat diese eine Auswertungsregel, da das einzige Nichtterminal *confdeflist* bereits seine Auswertungsregeln im darüber stehenden Block hat.
Der letzte auszuwertende Block ist der Hardwaremap Block. Er sieht den anderen Blöcken wieder sehr ähnlich.

```
hwblock: HARDWAREMAP '{' hwdeflist '}'
       ;

hwdeflist: hwclassdef { makeHwmapList(); }
         | hwclassdef hwdeflist { makeHwmapList(); }
 ;

hwclassdef: HWID ':' hwelid kommahwlistopt ';'
{ makeHwmap(); }
          ;


kommahwlistopt: kommahwlist
              | /* empty */
      ;

kommahwlist: ',' hwelid { makeList(); }
           | ',' hwelid kommahwlist { makeList(); }
   ;

hwelid: SENSID
      | POINTID
      | SIGID
      | CROSSID
      ;
```

Diese Auswertungsregeln sind in sehr ähnlicher Form allesamt weiter oben in diesem Abschnitt bereits behandelt worden.
Sind alle Blöcke reduziert worden, wird dann anhand der ersten Regel auf *tnd* reduziert.

```
 tnd: defblock coordblock sigpropblock ptpropblockopt slipswitchblockopt
     relblock sigposblock routeblock
condblock clearblock ptconfblockopt confblockopt hwblock
```

Hiermit wäre der Aufbau der Symboltabelle und des Syntaxbaums abgeschlossen. Die weiteren Schritte zur Generierung der Projektierungsdaten folgen in den nächsten Abschnitten.

7.4.4.3 Projektierungsdaten

In diesem Abschnitt soll erläutert werden wie die einzelnen Projektierungsdaten für den Route Dispatcher, den Route Controller und den Safety Monitor erzeugt werden, die der Steuerinterpreter zur effizienten Steuerung eines Gleisnetzes benötigt. Nähere Information befinden sich im Kapitel zum Steuerinterpreter 7.5.

7.4.4.3.1 Route Dispatcher Der Route Dispatcher dient im Steuerinterpreter dazu, nur Routen freizugeben, ide nicht mit zu dem Zeitpunkt freigegebenen Routen in Konflikt stehen. Hierzu benötigt dieser eine Konflikttabelle, die ihm der Compiler zur Verfügung stellt. Zusätzlich werden noch Daten für notwendige Warteschlangen bei mehreren Routenanforderungen generiert. Der Compiler benötigt für diese Erstellung die Daten aus dem Syntaxbaum und der Symboltabelle. Die Hauptfunktion zum Generieren der Route Dispatcher Projektierungsdaten ist

```
struct rd_data_t *getRDData(struct section_t *syntree);
```

Diese erhält als Eingabe den zum Gleisnetz gehörenden Syntaxbaum und erzeugt daraus die Projektierungsdaten für den Route Dispatcher. Diese Daten sind dann im folgenden Struct enthalten:

```
struct rd_data_t {
    int max_routes;
    char **rct;
    int max_queues;
    int *queue_tab;
};
```

Hierbei ist *maxroutes* die Anzahl der Maximalen Routen. *rct* beinhaltet die Zweidimensionale Konflikttabelle, die aussagt welche Route mit welcher anderen Route in einem Konflikt steht. Die Anzahl der Routerequest Queues ist in *maxqueues* enthalten und in *queuetab* liegt eine Zuordnung der Routen zu den Queues vor. Als erstes wird in dieser Funktion der Speicher für die Projektierungsdaten allokiert. Danach wird die Anzahl der Routen berechnet und in das Struct eingetragen. Als nächster Schritt wird über eine Schleife die zweidimensionale Routenkonflikttabelle allokiert. Danach werden die harten und weichen Konflikte in diese Tabelle durch den Aufruf der Funktion

```
void setConflicts(struct conflict_t *confs, struct rd_data_t *rdd);
```

eingetragen. Diese Funktion erhält eine Liste von Routenkonflikten und trägt für jeden Konflikt den entsprechenden Wert in die Konflikttabelle der Route-Dispatcher Projektierungsdaten ein. Diese ist als Zieladresse zu spezifizieren. Um dann die Menge der benötigten Queues zu bestimmen wird in einer Schleife über alle Routen folgende Funktion aufgerufen:

```
struct queue_tmp_t *makeQueueTmp(struct route_t *routedef,
 struct queue_tmp_t *list);
```

Wobei hier der Struct folgendermaßen aussieht:

```
struct queue_tmp_t{
    int queue;
    int rr;
    int rt;
    struct queue_tmp_t *next;
};
```

queue ist hier die ID des Queues, *rr* die ID des Routerequest Sensors, *rt* die ID der Route und *next* der Verweis auf das nächste Element der Liste. Die eben genannte Funktion erstellt eine Liste der erforderlichen Queues, um mehrere Routen auf einen RouteRequest Sensor abzubilden. Hierbei wird bei jedem Aufruf eine neue Routendefinition und die bis jetzt erstellte Queue Liste übergeben, die dann gegebenenfalls ergänzt wird.
Die Anzahl der Queues wird dann in die Route Dispatcher Projektierungsdaten eingetragen. Als nächstes wird die Tabelle für die Queues allokiert und über eine Schleife über alle Queues für jede Route ihre Queue eingetragen. Damit wären die Projektierungsdaten für den Route Dispatcher generiert.

7.4.4.3.2 Route Controller Der *TRACS* Steuerinterpreter hat für jede zu steuernde Route einen Route Controller und benötigt somit auch die Projektierungsdaten für die einzelnen Routen vom Compiler. In diesen Datan enthalten für jede Route binäre Listen von Steuerimpulsen für jegliche Elemente des Gleisnetzes um ein Schalten, Freigeben oder Sperren der Route zu ermöglichen. Die Route Controller Projektierungsdaten für einen Route Controller sehen folgendermaßen aus:

```
struct rc_data_t {
    int route_id;
    struct shm_value_t *free_sig_pos;
    struct shm_value_t *lockfirst_sig_pos;
    struct shm_value_t *point_pos;
    struct shm_value_t *lock_sig_pos;
```

```
    struct shm_value_t *req_sig_pos;
    int entry_sens_id;
    int exit_sens_id;
};
```

Wobei *routeid* die ID der entsprechen Route ist, *freesigpos* die Aktionsliste zum Freigeben der Route ist, *lockfirstsigpos* die Aktionslist zum Rotstellen des ersten Signals, *pointpos* die Liste zum Stellen der Weichen, *reqsigpos* die Liste zum Anschalten der Route Request Signale, *entrysensid* der Eingangssensor und *exitsensid* der Ausgangssensor der Route. Die Funktion zur Generierung dieser Daten für jeden einzelnen Route Controller ist folgende:

```
void getRCData(struct section_t *syntree,
       struct rc_data_t *newRcd,
       int rid);
```

Als Eingabedaten erhält diese Funktion, den Syntaxbaum des entsprechenden Gleisnetzes, die zu benutzende Zieladresse für die Route Dispatcher Projektierungsdaten und die ID der entsprechenden Route. Als erstes wird die Routen-ID eingetragen. Danach werden mit Hilfe der *clearances* aus dem Syntaxbaum die Signalstellung für die Route gesucht und der Wert für die *firstsigpos* eingetragen. Dies geschieht mit der Funktion

```
void setStopSignal(struct condclear_t *clear,
   struct shm_value_t *sig,
   int io);
```

Diese Funktion ermittelt anhand einer Signalbedingung einen entsprechenden SHM Wert, der je nach Steuerparameter in den entsprechenden Bereich der Zieladresse schreibt. Hierbei wird bei False der Eingabe- und Ausgabebereich gesetzt, bei True nur der Ausgabebereich. Dann wird mit einer Schleife über alle Signale mit der selben Funktion der Wert für die *locksigpos* eingetragen. Innerhalb dieser Schleife wird auch der Wert für die *freesigpos* ermittelt und eingetragen. Dies geschieht durch den Aufruf folgender Hilfsfunktion:

```
void setSignal(struct condclear_t *clear,
       struct shm_value_t *sig);
```

Als nächstes wird über eine Schleife aller Weichenstellungen für diese Route aus dem Syntaxbaum der Wert für die *pointpos* eingetragen. Hierzu wird folgende Hilfsfunktion genutzt:

```
void setPoint(struct condclear_t *cond,
      struct shm_value_t *pnt);
```

Danach wird aus dem Syntaxbaum die Sensorenliste für die Route erstellt. Mit der Funktion

```
int *getSensorList(struct idlist_t *senslist);
```

wird ein Feld von Integern erstellt, welches die SHM-IDs der entsprechenden Sensoren enthält. Jetzt wird der letzte Sensor der Route ermittelt und in das Struct eingetragen, ebenso wie der erste Sensor. Dann muß noch der Wert für die *reqsigpos* ermittelt und eingetragen werden. Dies wird über folgende Hilfsfunktion erreicht:

```
struct shm_value_t *setRRSignal(struct condclear_t *clear);
```

Damit wären dann die Projektierungsdaten für einen Route Controller generiert.

7.4.4.3.3 Safety Monitor In diesem Abschnitt soll erläutert werden wie die Projektierungsdaten für den Safety Monitor generiert werden. Der Steurinterpreter benötigt diese Daten, um zu jeden Zeitpunkt der Ausführung zu prüfen, ob sich das gesteuerte Gleisnetz in einem sicheren Zustand befindet. Dazu werden Sensorrelationen, Weichen- und Signalstellungen und Fehlermeldungen der Hardwareelemente ausgewertet. Die Projektierungsdaten sollen folgendermaßen aussehen:

```
struct sm_data_t {
        struct sm_condition_t **sm_conditions;
        struct hw_timeout_t *hw_timeouts;
};
```

Hier ist *smconditions* ein zweidimensionales Feld für die Sicherheitsbedingungen und *hwtimeouts* ein Feld für die Timerbedingungen.
Damit diese Daten generiert werden gibt es folgende Funktion:

```
struct sm_data_t *getSMData(struct section_t *syntree,
    struct rc_data_t *rcdata,
    struct rd_data_t *rddata);
```

Sie erhält als Eingaben den Syntaxbaum, die bereits generierten Projektierungsdaten für den Route Dispatcher und die Route Controller und generiert aus diesen die Projektierungsdaten für den Safety Monitor. Wie dies im einzelnen funktioniert wird jetzt beschrieben. Als erstes wird in dieser Funktion die Anzahl der Timer berechnet. Dann werden die sicheren Befahrungskonstellationen ermittelt. Dies geschieht mit folgender Hilfsfunktion:

```
struct occ_conf_t *getSafeOccConfs(struct rd_data_t *rdd);
```

Als Eingabe erhält sie die Route Dispatcher Projektierungsdaten. Hier werden zunächst alle Befahrungskonstellationen ermittelt die theoretsich möglich sind. Mit der Hilfsfunktion

```
int isValidCombo(u_int32_t combo,
 struct rd_data_t *rdd);
```

werden dann alle Konstellationen einzeln überprüft, ob sie auch sicher sind. Diese sicheren Befahrungskonstellationen werden dann wieder zurückgegeben und mit der Funktion

```
struct sm_condition_t **getSmConditionByOcc(struct occ_conf_t *occ,
    struct rc_data_t *rcdata,
    struct section_t *syntree);
```

werden die Projektierungsdaten für den Safety Monitor anhand der sicheren Befahrungskonstellationen erzeugt, indem für jeden Zustand alle Zustände der einzelnen Hardwareelemente eingetragen werden.
Dann werden über mehrere Schleifen über alle Hardwaretypen die zugehörigen Timeouts erstellt und eingetragen. Dies geschieht mit folgender Hilfsfunktion:

```
struct hw_timeout_t *getHwTimeout(struct hw_timeout_t *newhw,
  int stime, int shmid);
```

Hiermit sind dann die Projektierungsdaten für den Safety Monitor erzeugt.

Waldemar Wockenfuß

7.4.5 Schnittstellen

In diesem Abschnitt sollen die Schnittstellen des *tram* Compilers erläutert werden. Dabei sollen die externen als auch die wichtigsten internen Schnittstellen betrachtet werden. Wobei die externen Schnittstellen in den vorherigen Abschnitten schon genauer betrachtet wurden und hier nur noch einmal kurz erläutert werden. Als erste externe Schnittstelle benötigt der Compiler Eingabedaten. Die zweit externe Schnittstelle sind die Projektierungsdaten für den Steuerinterpreter. Als interne Schnittstellen sollen die zentralen Funktionen des Compilers betrachtet werden.

7.4.5.1 Eingabedaten

Die Eingabedaten die der *tram* Compiler erhält bestehen aus einer Textdatei in Form der *TND* Beschreibung die im Anhang
genauer erklärt wird. In Abschnitt 7.4.4 wird genauer auf das Zusammenspiel zwischen Compiler und TND eingegangen. Im Prinzip wir die Originale EBNF Beschreibung genommen, die in dem Compiler in eine BNF gewandelt wurde. Diese wird dann geprüft und deren Daten in einen Syntaxbaum und eine Symboltabelle geschrieben, die das weitere Verarbeiten der Daten ermöglichen.

7.4.5.2 Zentrale Funktionen

Die internen Schnittstellen des Compilers sind seine eigenen Funktionen. In den vorherigen Abschnitten wurde bereits gezeigt, wie der Compiler aus den eingelesenen Daten den Syntaxbaum erstellt. Die Funktionen um die Projektierungsdaten zu erstellen wurden im Abschnitt davor behandelt, sollen aber hier noch einmal kurz erläutert werden. Als erstes gibt es

```
void getRCData(struct section_t *syntree,
       struct rc_data_t *newRcd,
       int rid);
```

Diese Funktion analysiert den gegebenen Syntaxbaum und erstellt für eine per ID gegebene Route die benötigten Aktionslisten für den Steuerinterpreter. Hierbei werden mithilfe anderer Funktionen einzelne Informationen aus dem Syntaxbaum in einzelne SHM Steuerbefehle umgesetzt. Die Ausgabe erfolgt in einen spezifizierten Zielbereich, der bereits allokiert sein muß. Als Eingaben erhält die Funktion den zu benutzenden Syntaxbaum, die zu benutzende Zieladresse und die jeweils zu bearbeitende Route. Diese Funktion erstellt die Projektierungsdaten für alle Route Controller.
Als nächstes gibst es da:

```
struct rd_data_t *getRDData(struct section_t *syntree);
```

Hier erhält die Funktion nur den Syntaxbaum mitsamt seiner Daten und erstellt daraus eine Konflikttabelle und die notwendigen Route Request Queues.
Dann gibt es noch:

```
struct sm_data_t *getSMData(struct section_t *syntree,
    struct rc_data_t *rcdata,
    struct rd_data_t *rddata);
```

Hier werden die Daten für den Safety Monitor erzeugt. Diese Funktion erhält die bereits generierten Daten für den Route Dispatcher und für die Route Controller und den Syntaxbaum und liefert die Projektierungsdaten für den Safety Monitor. Wie dies im einzelnen funktioniert ist im Abschnitt zuvor besachrieben worden.
Die Funktion

```
struct projdata_t *getProjData(struct section_t *syntree);
```

ist ein Wrapper für die drei zuvor genannten Funktionen. Sie ruft sie alle auf, um die einzelnen Projektierungsdaten zu generieren und gibt sie dann gemainsam wieder.

7.4.5.3 Ausgabedaten

Als zweite externe Schnittstelle dienen die Projektierungsdaten. Wie diese aussehen sollen und wie sie erzeugt werden ist bereits im Abschnitt 7.4.4 abgearbeitet worden. Eine genauere Beschreibung der Projektierungsdaten findet sich im Kapitel über den Steuerinterpreter.

Waldemar Wockenfuß

7.4.6 Systemvoraussetzungen

Für den TRACS *tram* Compiler gelten folgende Systemvoraussetzungen:

- Linux Betriebssystem (Getestet wurde *SuSE Linux 9.0 Professional*)
- Standardbibliotheken
  - `<stdio.h>`
  - `<stdlib.h>`
  - `<search.h>`
- Lexergenerator `flex` ab Version 2.5.4
- Parsergenerator `bison` ab Version 1.75
- Buildwerkzeug `make` ab Version 3.80
- C Compiler/Linker `gcc` ab Version 3.3.1

Helge Löding

7.4.7 Bedienung

Der TRACS *tram* Compiler wird durch folgenden Aufruf gestartet:

```
tram <EINGABEDATEI> [<AUSGABEDATEI>]
```

Helge Löding & Waldemar Wockenfuß

7.4.8 Reflexion

In diesem Abschnitt soll wiedergegeben werden wie unsere Arbeit an dem *tram* Compiler verlief. Hierbei soll auf positives wie auch auf negatives bei der Fertigstellung eingegangen werden. Mit unserem Endprodukt sind wir sehr zufrieden, es erledigt genau die Arbeiten die auch vorgesehen waren. Allerdings gab es auch Probleme. Früh in der Projektphase haben wir einen Netzplan für unseren Bereich geschaffen, um ein Vorbild für die anderen Gruppen zu sein. Dies klappte allerdings aus zwei Gründen nicht. Zuerst haben die anderen Gruppen nicht sofort nachgezogen und zum zweiten lagen wir in unserer Planung komplett falsch. Anfangs sind wir davon ausgegangen, dass wir den Compiler in einem Semester fertigstellen, was uns aber absolut nicht möglich war. Wir haben jegliche Zeiteinschätzungen wegen der Einarbeitungszeiten in jeden einzelnen Bereich unterschätzt. Dazu kamen immer wieder Änderungen, die im Nachhinein eingearbeitet werden mußten. Desweiteren wurde von Anfang an dir Größe der Gruppe falsch eingeschätzt. Anfangs gab es noch drei Mitglieder, wobei eines noch früh im ersten Semester verloren ging. Später sollte diese Gruppe wieder auf drei Mann aufgestockt werden, was allerdings aufgrund der Motivation des Neuen auch nicht klappte. So blieb diese Gruppe im Endeffekt immer zwei Mann stark. Durch gravierende persönliche Probleme um das zweite Semester herum konnte auch einer dieser zwei nicht besonders viel zur Arbeit beitragen, so dass die Arbeitskraft auf tatsächlich einen und einen halben Mann bergrenzt war. Diese Situation hat sich wieder gefangen und es lief dann auch wieder besser, aber durch ständige Änderungen, war nur selten ein Ziel zu sehen. Am Ende ist aber dennoch das Ziel erreicht worden mit der Zufriedenheit der Gruppe. Die Zusammenarbeit innerhalb der Gruppe verlief vorbildlich, auch wenn nicht schwer mit zwei Mann. Die Zusammenarbeit mit den anderen Gruppen war durchaus erfolgreich. Insbesondere mit den Leuten vom Steuerinterpreter wurde untereinander viel ausgetauscht. Immer wenn man einen Ansprechpartner einer anderen Gruppe gebraucht hat, hat sich auch jemand gemeldet. Als Fazit bleibt zu sagen, daß man viel gelernt hat, insbesondere was die Zusammenarbeit mit Anderen angeht, meistens funktioniert dies, aber leider manchmal auch nicht. Mit unserem Ergebnis sind wir jedenfalls zufrieden.

Waldemar Wockenfuß

7.5 Steuerinterpreter

7.5.1 Überblick

Der Steuerinterpreter ist eine zentrale Komponente des Projektes, die den generischen Teil der Steuerungssoftware bildet und als solches das Gleisnetz steuern muss. An ihn werden folgende Anforderungen bezüglich der Funktionalität gestellt:

- Leiten von Zügen durch einen Gleisnetzabschnitt
- Kommunikation mit den Gleisnetzelementen (Weichen, Sensor, Signal)
- Reaktion auf Sicherheitsverletzungen (defekte Hardware, unsichere Freigaben von Routen)
- Vermeidung der im FTA (siehe Abschnitt 8.2) gefundenen potenziellen Bedrohungen
- Universelle Verwendbarkeit für verschiedenste Gleisnetze

Außerdem werden folgende strukturelle Anforderungen gestellt:

- Einfacher Aufbau, um zur Vermeidung von Compilervalidierung eine Implementierung in Assembler zu ermöglichen (was aber im Projekt nicht mehr realisiert wurde).
- Verwendung von Standard-PC-Hardware aus Kostengründen
- Einfache Verifizier- und Testbarkeit

Mit der nachfolgend beschriebenen Architektur sollen diese Anforderungen erfüllen werden. Die darauffolgende Spezifizikation der Schnittstelle zu den Treibern, der Systemumgebung und der einzelnen Teilkomponenten muss präzise genug sein, damit eine Verifizier- und Testbarkeit erreicht wird.

Deng Zhou, Andreas Kemnade

7.5.2 Architektur

In den folgenden Unterabschnitten wird die Architektur der einzelnen Teile des Steuersystems dargestellt. Wir haben diese Architektur so gewählt, dass die einzelnen Teile unseres Systems eine klare Aufgabenteilung haben. Dabei haben wir uns an dem Safety Architecture Framework aus der SCS1-Vorlesung orientiert [PZ]. Dabei entspricht unser Treiberframework dem System interface layer und der Safety Monitor entspricht dem Safety Layer. Route Controller und Route Dispatcher entsprechen dabei etwas dem User Interface Layer, da hier Routenanforderungen von der Bahn/dem Fahrer (also dem Benutzer) verarbeitet werden.
Das Treiberframework dient dazu, die Treiber zu verwalten.

7.5.2.1 Gesamtsystem

Die eigentliche Steuerungssoftware besteht aus zwei grossen Blöcken, die jeweils in eigenen Prozessen ablaufen. Zum einen gibt es den Treiberblock, der für die Hardwaresteuerung zuständig ist. Zum anderen haben wir den Safety Controller, der den Route Dispatcher, die Route Controller und den Safety Monitor beinhaltet.
Die Anpassung der Steuerungssoftware an das konkrete Gleisnetz erfolgt durch Konfigurationsdaten. Diese Konfigurationsdaten beinhalten sowohl eine Beschreibung sämtlicher Hardwareelemente als auch alle notwendigen Informationen, wie die vorhandenen Routen zu steuern sind. Die Daten bezüglich der Zuordnung von Hardware-Elementen zu den Treibern werden in Headerdateien geschrieben, die damit direkt beim Kompilieren des Treiberframeworks verarbeitet werden.
Der Rest wird jedoch vom TRACS Compiler (siehe Abschnitt 7.4) in das spezifizierte Projektierungsdatenformat übersetzt und zur Laufzeit von der Steuerungssoftware geladen. Die Projektierungsdatendatei wird dem Steuerinterpreter als Kommandozeilenparameter beim Start übergeben.

7.5.2.1.1 Programmierstrategien

Um gegen Programmfehler vorzusorgen, soll folgendes nicht geschehen: Es soll während des Programmablaufs kein Speicher dynamisch alloziert werden, sondern nur während der Initialisierung. Strings sollten möglichst nicht verwendet werden, damit es keine Probleme mit zu langen Strings gibt. Werden Arrays verwendet, so sind die Grenzen zu überprüfen. Schleifen (`for` und `while`) sind so zu konstruieren, dass sie eine garantierte Maximalanzahl an Durchläufen haben, es sind also Zähler zu verwenden. Eine Ausnahme darf nur die Hauptschleife sein, die nie beendet werden soll.
Diese Vorgaben sind bei der Entwicklung berücksichtigt worden. Schleifen sind stets mit Zählern versehen, die innerhalb der Schleife nicht verändert werden. In einem Fall wird allerdings der Zähler unter bestimmten Bedingungen zurückgezählt, aber da ist auf andere Art und Weise sichergestellt, dass die Schleife abbricht.

7.5.2.2 Treiberblock

Das Treiberframework ist zuständig für die Verwaltung sämtlicher Treiber, die für das zu steuernde Gleisnetz notwendig sind. Alle Treiber definieren dazu einen Satz von *Callback-Funktionen* (also treiberspezifische Funktionen die zu einem festgelegten Zeitpunkt aufgerufen werden), die vom Framework zur Laufzeit aufgerufen werden. Es sind Callbacks vorgesehen für die Initialisierung, einen Sollzustandswechsel sowie optional für eine allgemeine Ausführung. Sämtliche Treiberinformationen müssen zur Compilezeit vorliegen, da sie entscheiden, welche Objektdateien ins Framework gelinkt und welche Datenstrukturen generiert werden müssen. Zur Laufzeit initialisiert das Framework zunächst alle Treiber sowie seine eigenen globalen Datenstrukturen und die Shared-

Memory-Schnittstelle. Das Shared Memory dient dazu, um über Prozessgrenzen hinweg einen gemeinsam verwendbaren Speicherbereich zu haben. Jeder Treiber hat einen eigenen Shared-Memory-Bereich, von dem ein Teil der Sollbereich ist und ein Teil der Ist-Bereich. Genauere Informationen zu dieser Schnittstelle befinden sich im Abschnitt 7.5.3.1.
Sobald alle Initialisierungen abgeschlossen sind, werden in einer Endlosschleife für jeden aktiven Treiber die in der Spezifikation beschriebenen Callback-Funktionen aufgerufen Es wird also für jeden Treiber folgendes nacheinander durchgeführt:

- die SHM-Sollwerte ausgelesen und mit den vorhergehenden Werten verglichen
- bei Abweichungen wird die Callback-Funktion für Sollzustandswechsel aufgerufen
- wenn vorhanden wird die allgemeine Ausführungsfunktion aufgerufen
- bei Änderungen werden die SHM-Istwerte geschrieben
- Timer werden überprüft, ob sie gestartet wurden oder abgelaufen sind
- Sensorzähler werden verwaltet

7.5.2.3 Safety Controller

In diesem Abschnitt wird der Aufbau der verschiedenen Komponenten des Safety Controller beschrieben.
Der Safety Controller enthält im Wesentlichen drei Komponenten: Den Route Dispatcher (RD), eine Liste von Route Controllern (RC) und den Safety Monitor (SM). Der RD nimmt Route Requests von den Route Request-Sensoren entgegen, prüft ob sie nach den entsprechenden Abschnitten der Projektierungsdaten aktiviert werden können und markieren diese als aktiv. Die einzelnen Route Controller überprüfen jeweils, ob die entsprechende Route als aktiv markiert ist. Falls das der Fall ist, sorgen sie dafür, dass alle notwendigen Hardwareelemente entsprechend geschaltet werden, damit die Route befahren werden kann. Dies beinhaltet auch das finale Freigabesignal. Sie sind auch dafür zuständig die Signale entsprechend zu setzen, so dass die Route wieder gesperrt ist. Route wieder frei ist. Der Safety Monitor schliesslich vergleicht die von RD und RC angeforderten Zustände und den aktuellen Ist-Zustand mit einer Liste von Sicherheitsbedingungen und „schlägt Alarm“ falls diese verletzt werden. RD, RC und SM sind dabei jeweils Funktionen, die innerhalb einer Endlosschleife wie folgt aufgerufen werden:

- zuerst wertet der RD alle anliegenden Route Requests aus
- anschliessend werden die gerade aktiven RC aufgerufen, Änderungen werden jedoch nicht direkt an das Treiberframework gesendet sondern in einer Kopie des SHM gespeichert.

- Der SM überprüft schliesslich diese SHM-Kopie UND den aktuellen Ist-Zustand gegen die Sicherheitsbedingungen, falls keine Sicherheitsverletzungen vorliegen wird die Kopie and das Treiberframework gesendet. Andernfalls wird in einen Fehlerzustand gewechselt, der das System sichert und kontrolliert herunterfährt.

Marius Mauch, Andreas Kemnade, Deng Zhou

7.5.3 Schnittstellen

7.5.3.1 Shared Memory Zustände

Im folgenden wird zunächst allgemein der Aufbau der Shared-Memory-Schnittstelle erläutert, dann die Verteilung der Shared-Memory-IDs nach Hardwareklassen. Danach erfolgt eine Auflistung der einzelnen Bits für die jeweiligen Hardwareklassen getrennt nach Bits im Eingabe- und Ausgabebereich.
Jeder Treiber kommuniziert mit dem Steuerinterpreter (SI) über einen eigenen 32 Bit großen Speicherbereich. Dieser ist unterteilt in zwei je 16 Bit große Teile, wobei der eine Bereich vom Treiber gelesen und vom Steuerinterpreter geschrieben wird (Eingabebereich) und der andere vom Treiber geschrieben und SI gelesen wird (Ausgabebereich). Das bedeutet, dass auf diesen Speicherbereich unter Umständen von mehreren Prozessen zugegriffen werden muss, es sich dabei also um Shared Memory handelt. Durch diese einheitliche Schnittstelle wird die Treiber- und die SI-Entwicklung vereinfacht. Die Semantik für Ein- und Ausgabebereich ist von den einzelnen Treibern abhängig und wird in den folgenden Treiberspezifikationen definiert.
Prinzipiell würden für Ein- und Ausgabe auch je 8 Bit ausreichen, dies würde jedoch die Möglichkeiten für zukünftige Erweiterungen unnötig einschränken. Aufgrund der Vorgabe eines 32 Bit Systems würden des Weiteren keine Vorteile aus einer Beschränkung auf 8 Bit entstehen.
Alle Lese- und Schreibzugriffe auf einen einzelnen Speicherbereich müssen atomar sein. Sämtliche nicht definierten Bits müssen beim Auslesen ignoriert werden, es darf nicht davon ausgegangen werden, dass diese einen konstanten Wert haben. Jeder Treiber und der SI müssen den Speicherbereich kennen, der zur Kommunikation benutzt wird. Konkret heißt das, dass jeder Speicherbereich eine eindeutige ID bekommt, die der SI und der zugeordnete Treiber kennen.

7.5.3.1.1 Schema zu SHM-ID-Vergabe

allgemeiner Bereich	1000-9999	
Weichen:	1000-1999	
Kreuzungen:	2000-2999	
Signale:	3000-3999	
RR-Sensoren:	40xx	wobei xx die Routen-ID ist
Sensoren allgemein:	4000-4999	
Timer:	5000-7999	
Routen-Freigabe-Timer:	80xx	xx ist die Routen-ID
Routen-Sperr-Timer:	81xx	xx ist die Routen-ID

7.5.3.1.2 Weichen

Diese Spezifikation gilt für folgende Arten von Weichen wie im Weltmodell (Abschnitt 2.3) beschrieben, die sich durch die möglichen Schaltstellungen unterscheiden, welche in der folgenden Auflistung angegeben sind:

- straight-left (SL)
- straight-right (SR)
- left-right (LR)
- straight-left-right (SLR)

Eingabebereich

Der Eingabebereich für einen Weichen-Treiber beschreibt den gerade vom Steuerinterpreter angeforderten Zustand. Hierbei wird jede mögliche Anforderung binär codiert, indem jedem einzelnen Bit eine Bedeutung zugewiesen wird. Wenn im (definierten) Eingabebereich mehr als ein Bit gesetzt ist, liegt ein Fehler des Steuerinterpreters oder der Hardware vor und der Treiber muss in einen Fehlerzustand wechseln (s.u.).
Die folgenden Bits sind für den Eingabebereich definiert:

Bit 0:	Anforderung für GO-STRAIGHT Position
Bit 1:	Anforderung für GO-LEFT Position
Bit 2:	Anforderung für GO-RIGHT Position
Bit 15:	Anforderung in einen Fehlerzustand zu wechseln und die Ausführung zu beenden

Die Bits 3-14 sind nicht definiert und müssen vom Treiber ignoriert werden.

Ausgabebereich

Analog zum Eingabebereich signalisiert der Wert im Ausgabebereich den aktuellen Status der vom Treiber gesteuerten Weiche. Ähnlich wie beim Eingabebereich darf auch im (definierten) Ausgabebereich prinzipiell nur ein Bit gesetzt sein, allerdings ist es möglich sowohl einen Positionszustand als auch einen Fehlerzustand gleichzeitig abzubilden, in diesem Fall sind zwei gesetzte Bits erlaubt.
Die folgenden Bits sind für den Ausgabebereich definiert:

Bit 0: aktuelle Position ist STRAIGHT-POS
Bit 1: aktuelle Position ist LEFT-POS
Bit 2: aktuelle Position ist RIGHT-POS
Bit 13: Fehlerzustand OFF
Bit 14: Fehlerzustand INVALID
Bit 15: Fehlerzustand FAILURE

Die Bits 3-12 sind nicht definiert und müssen vom Steuerinterpreter ignoriert werden.
Die verschiedenen Fehlerzustände haben folgende Bedeutungen:

OFF: Die Ausführung des Treibers wurde beendet, es werden keine Anforderungen mehr ausgeführt.
INVALID: Die letzte Anforderung war ungültig, z.B. weil Bit 1 und Bit 2 gleichzeitig gesetzt waren.
FAILURE: Ein Hardwarefehler wurde erkannt.

7.5.3.1.3 Kreuzungsweichen/Slipswitches

Diese Spezifikation gilt für aktive Weichen, die aus zwei bzw. vier 2-Wege Weichen bestehen, die in Form einer Kreuzung angeordnet sind. Diese Kreuzungen werden in der TND als Slipswitches bezeichnet.

Eingabebereich

Der Eingabebereich für einen Slipswitch-Treiber beschreibt den gerade vom Steuerinterpreter angeforderten Zustand. Hierbei wird jede mögliche Anforderung binär codiert, indem jedem einzelnen Bit eine Bedeutung zugewiesen wird. Wenn im (definierten) Eingabebereich mehr als ein Bit gesetzt ist liegt ein Fehler des Steuerinterpreters oder der Hardware vor und der Treiber muss in einen Fehlerzustand wechseln (s.u.). Die folgenden Bits sind für den Eingabebereich definiert:

Bit 0: Anforderung für GO-STRAIGHT Position
Bit 1: Anforderung für GO-BENT Position
Bit 15: Anforderung in einen Fehlerzustand zu wechseln und die Ausführung zu beenden

Die Bits 2-14 sind nicht definiert und müssen vom Treiber ignoriert werden.

Ausgabebereich

Analog zum Eingabebereich signalisiert der Wert im Ausgabebereich den aktuellen Status der vom Treiber gesteuerten Kreuzung. Ähnlich wie beim Eingabebereich darf auch im (definierten) Ausgabebereich prinzipiell nur ein Bit gesetzt sein, allerdings ist es möglich sowohl einen Positionszustand als auch einen Fehlerzustand gleichzeitig abzubilden, in diesem Fall sind zwei gesetzte Bits erlaubt. Die folgenden Bits sind für den Ausgabebereich definiert:

Bit 0: aktuelle Position ist STRAIGHT-POS
Bit 1: aktuelle Position ist BENT
Bit 13: Fehlerzustand OFF
Bit 14: Fehlerzustand INVALID
Bit 15: Fehlerzustand FAILURE

Die Bits 2-12 sind nicht definiert und müssen vom Steuerinterpreter ignoriert werden. Die verschiedenen Fehlerzustände haben folgende Bedeutungen:

OFF: Die Ausführung des Treibers wurde beendet, es werden keine Anforderungen mehr ausgeführt.
INVALID: Die letzte Anforderung war ungültig, z.B. weil Bit 1 und Bit 2 gleichzeitig gesetzt waren.
FAILURE: Ein Hardwarefehler wurde erkannt.

Der Steuerinterpreter muss auf alle Fehlerzustände gleich reagieren und den betroffenen Abschnitt gemäss den Sicherheitsbedingungen absperren.

7.5.3.1.4 Sensoren

Diese Spezifikation gilt für folgende Arten von Sensoren wie im Weltmodell (Abschnitt 2.3) beschrieben:

- Route-Request (RR)
- Toggle mit Richtung (TD)
- Toggle ohne Richtung (TG)
- Zustand mit Richtung (SD)
- Zustand ohne Richtung (SG)

Hierbei existiert für jede Route ein RR-Sensor der immer auslöst, sobald die ihm zugeordnete Route angefordert wird. Wenn ein physikalischer Route Request Sensor mehrere Routen auslösen kann, dann muss es dafür für jede Route einen logischen RR-Sensor geben.
Toggle-Sensoren schalten bei jedem Kontakt ihr Toggle-Bit um, sie können jedoch nicht die aktuelle Anwesenheit eines Zugs überprüfen. Zustandssensoren können dies.
Hat der Sensor eine Richtungserkennung, so steht zusätzlich die Information bereit, in welcher Richtung der Sensor überquert wurde

Eingabebereich

Der Eingabebereich für einen Sensor-Treiber beschreibt den gerade vom Steuerinterpreter angeforderten Zustand. Hierbei wird jede mögliche Anforderung binär codiert, indem jedem einzelnen Bit eine Bedeutung zugewiesen wird. Die folgenden Bits sind für den Eingabebereich definiert:

Bit 0: Alle Sensorwerte zurücksetzen (hauptsächlich für RR)
Bit 15: Anforderung in einen Fehlerzustand zu wechseln und die Ausführung zu beenden

Die Bits 1-14 sind nicht definiert und müssen vom Treiber ignoriert werden.

Ausgabebereich

Analog zum Eingabebereich signalisiert der Wert im Ausgabebereich den aktuellen Status des vom Treiber gesteuerten Sensors. Hierbei dürfen Bit 0 und 1 nicht gleichzeitig gesetzt sein, ebenso darf ein Treiber nur Bit 2 oder Bit 3 benutzen. RR-Sensoren dürfen nur Bit 10 benutzen. Die folgenden Bits sind für den Ausgabebereich definiert:

Bit 0: Richtung A
Bit 1: Richtung B
Bit 2: Zug ist anwesend
Bit 3: Toggle
Bit 10: Route-Request ausgelöst
Bit 13: Fehlerzustand OFF
Bit 15: Fehlerzustand FAILURE

Die Bits 4-9, 11-12 und 14 sind nicht definiert und müssen vom Steuerinterpreter ignoriert werden. Die verschiedenen Fehlerzustände haben folgende Bedeutungen:

OFF: Die Ausführung des Treibers wurde beendet, es werden keine Anforderungen mehr ausgeführt.
FAILURE: Ein Hardwarefehler wurde erkannt

7.5.3.1.5 Signale

Diese Spezifikation gilt für folgende Arten von Signalen wie im Weltmodell (Abschnitt 2.3) beschrieben:

- straight-left (SL)
- straight-right (SR)
- left-right (LR)
- straight-left-right (SLR)
- straight (S)
- left (L)
- right (R)

Eingabebereich

Der Eingabebereich für einen Signal-Treiber beschreibt den gerade vom Steuerinterpreter angeforderten Zustand. Hierbei wird jede mögliche Anforderung binär codiert, indem jedem einzelnen Bit eine Bedeutung zugewiesen wird. Wenn im (definierten) Eingabebereich mehr als ein Bit gesetzt ist liegt ein Fehler des Steuerinterpreters oder der Hardware vor und der Treiber muss in einen Fehlerzustand wechseln (s.u.), eine Ausnahme sind REQUEST RECEIVED Anzeigen, die zusätzlich zu GO/STOP/WAIT Anzeigen geschaltet werden können. Die folgenden Bits sind für den Eingabebereich definiert:

Bit 0:	Anforderung für GO-STRAIGHT Anzeige
Bit 1:	Anforderung für GO-LEFT Anzeige
Bit 2:	Anforderung für GO-RIGHT Anzeige
Bit 3:	Anforderung für STOP Anzeige
Bit 4:	Anforderung für REQUEST RECEIVED STRAIGHT Anzeige
Bit 5:	Anforderung für REQUEST RECEIVED LEFT Anzeige
Bit 6:	Anforderung für REQUEST RECEIVED RIGHT Anzeige
Bit 7:	Anforderung für WAIT STRAIGHT Anzeige
Bit 8:	Anforderung für WAIT LEFT Anzeige
Bit 9:	Anforderung für WAIT RIGHT Anzeige
Bit 15:	Anforderung in einen Fehlerzustand zu wechseln und die Ausführung zu beenden

Die Bits 10-14 sind nicht definiert und müssen vom Treiber ignoriert werden.

Ausgabebereich

Analog zum Eingabebereich signalisiert der Wert im Ausgabebereich den aktuellen Status des vom Treiber gesteuerten Signals. Ähnlich wie beim Eingabebereich darf auch im (definierten) Ausgabebereich prinzipiell nur ein Bit gesetzt sein (Ausnahme für REQUEST RECEIVED gilt auch hier), allerdings ist es möglich sowohl einen Anzeigezustand als auch einen Fehlerzustand gleichzeitig abzubilden, in diesem Fall sind zwei (bzw. drei) gesetzte Bits erlaubt. Die folgenden Bits sind für den Ausgabebereich definiert:

Bit 0: aktuelle Anzeige ist STRAIGHT-POS
Bit 1: aktuelle Anzeige ist LEFT-POS
Bit 2: aktuelle Anzeige ist RIGHT-POS
Bit 3: aktuelle Anzeige ist STOP
Bit 4: aktuelle Anzeige ist REQUEST RECEIVED STRAIGHT
Bit 5: aktuelle Anzeige ist REQUEST RECEIVED LEFT
Bit 6: aktuelle Anzeige ist REQUEST RECEIVED RIGHT
Bit 7: aktuelle Anzeige ist WAIT STRAIGHT
Bit 8: aktuelle Anzeige ist WAIT LEFT
Bit 9: aktuelle Anzeige ist WAIT RIGHT
Bit 13: Fehlerzustand OFF
Bit 14: Fehlerzustand INVALID
Bit 15: Fehlerzustand FAILURE

Die Bits 10-12 sind nicht definiert und müssen vom Steuerinterpreter ignoriert werden.

OFF: Die Ausführung des Treibers wurde beendet, es werden keine Anforderungen mehr ausgeführt.
INVALID: Die letzte Anforderung war ungültig, z.B. weil Bit 1 und Bit 2 gleichzeitig gesetzt waren.
FAILURE: Ein Hardwarefehler wurde erkannt

Anmerkungen

Da Signale der Typen SL, SR und LR nur je zwei Zustände kennen, ist der jeweils dritte Zustand, der nicht der Hardware entspricht, als nicht definiert anzusehen und bei einer entsprechenden Anforderung muss der Treiber in einen Fehlerzustand wechseln.

7.5.3.1.6 Timer

Diese Spezifikation gilt für Timer mit festgelegter Laufzeit.

Eingabebereich

Der Eingabebereich für einen Timer dient dazu, ihn zu starten und dabei zurückzusetzen.

Bit 0:	Anforderung für Timer starten
Bit 15:	Anforderung in einen Fehlerzustand zu wechseln und die Ausführung zu beenden

Die Bits 1-14 sind nicht definiert und müssen vom Treiber ignoriert werden.

Ausgabebereich

Der Ausgabebereich gibt den aktuellen Zustand des Treibers an.

Bit 0:	Timer läuft
Bit 1:	Timer ist abgelaufen
Bit 15:	Fehlerzustand OFF

Die Bits 2-14 sind nicht definiert und müssen vom Steuerinterpreter ignoriert werden.
Der Timer hat initial kein Bit gesetzt. Sobald das Startbit gesetzt ist, wird der Timer gestartet und Bit 0 gesetzt und das Bit 0 im Eingabebereich gelöscht. Ist der Timer abgelaufen, wird Bit 0 gelöscht und Bit 1 gesetzt.
Diese Timer werden intern für den SI im Treiberframework verwaltet. Durch die zur Compilierzeit vorliegenden Informationen wird die Laufzeit festgelegt.

7.5.3.2 Verbindung zur Hardware

In diesem Abschnitt geht es darum, ein paar Möglichkeiten darzustellen, wie die Anforderungen von der Hardware zur Shared-Memory-Schnittstelle gelangen können. Das Problem ist, dass kaum Informationen zu den Schnittstellen zu konkreter Gleishardware gibt, Entsprechende Fragen sind Bestandteil der Frageliste an Hanning und Kahl gewesen. Diese Frageliste wurde aber nicht beantwortet. Dadurch stehen die notwendigen Informationen nicht zur Verfügung, um den obengenannten Weg vollständig zu beschreiben. Innerhalb des Projektes erfolgten daher keine weiteren Arbeiten an diesem Weg ausser dieser Beschreibung. Es wurde jedoch schon deutlich, dass bei Hanning und Kahl ausserhalb des Schaltschranks keine Digitaltechnik eingesetzt wird, wobei Hanning und Kahl ihr System (Schaltschrank und Bahntechnik) nur komplett verkaufen. Eine weitere

Frage ist, wieweit „Intelligenz“ auf den Schnittstellenkarten, die bei Hanning und Kahl verwendet werden, z.B. in Form von kleinen, billigen Mikrocontrollern vorhanden ist, so dass dort sich auch Software befindet, die dem Treiber Arbeit abnehmen kann. Fehlt diese „Intelligenz“ und müssen dadurch serielle Protokolle bitweise generiert werden, dann können schwierige Timingprobleme im Mikrosekundenbereich entstehen.
Im folgendes werden exemplarisch ein paar Möglichkeiten gezeigt, wie man digitale Ein-/Ausgänge erhalten kann, an die man z.B. Leuchtdioden und Taster anschließen kann. Das wäre immerhin eine Annäherung an den realen Einsatz, da man dann wenigsten überhaupt mal Hardware hätte, um sie anzusteuern. Diese Möglichkeiten wurden im Projekt nicht ausprobiert. Um reale Weichen anzusteuern, bräuchte man noch weitere Elektronik.

7.5.3.2.1 Beispiel für eine PCI-IO-Karte

Als Beispiel für eine PCI-IO-Karte wird die Karte mit der Artikelnummer ADAP37 von Decision Computer vorgestellt. [Dec] Diese Karte hat 48 I/O-Leitungen und mehrere Timer.

Ansteuerung

Die I/O-Leitungen können angesteuert werden, indem auf die entsprechenden I/O-Register der Karte zugegriffen wird. Die zwei Chips mit der Bezeichnung 8255 haben vier Register, wovon eins zur Konfiguration benutzt wird, man also einstellt, welche Leitungen als Eingang und welche als Ausgang benutzt werden sollen. Die anderen drei Register dienen dann dazu die Leitungen anzusteuern. (([Ins92, S. 268ff]) Auf diese Register kann mit den C-Makros `inb()` und `outb()` zugegriffen wird, wenn geeignete Zugriffsrechte vorhanden sind. Das bedeutet, dass man, wenn es ausserhalb des Kernels funktionieren soll, die entsprechenden IO-Bereiche mit `ioperm()` freischalten muss.

7.5.3.2.2 Beispiel für eine USB-IO-Karte

Als Beispiel für eine Möglichkeit, durch den USB-Ausgang I/O-Leitungen zu bekommen, besteht, dazu die Bausteine mit der Bezeichnung IOWarrior der Firm Code Mercenaries zu benutzen [Cod]. Sie haben bis zu 32 I/O-Leitungen, dabei gibt es spezielle Unterstützung für einige serielle Protokolle.

Ansteuerung

Die Ansteuerung kann durch die libusb (ist als Paket bei den meisten Linux-Distributionen enthalten) erfolgen oder durch ein Kernelmodul dafür.

7.5.4 Spezifikation

In den folgenden Abschnitten werden die Eingaben, die Ausgaben und das Verhalten der einzelnen Teile des Systems bestehend aus Treiberblock und Safety Controller definiert. Dabei ist zu beachten, dass stets definiert wird, was in einer Iteration geschieht. Sicherheitskritische Echtzeitanforderungen tauchen dort nicht auf. Es gibt aber sehr wohl Zeiten, die das Komfortverhalten beeinträchtigen, etwa in der Form, dass Bahnen zu lange auf die Freigabe von Routen warten müssen, oder bremsen müssen. Wenn Timer verwendet werden (also zur Defekterkennung bei Überschreitung der maximalen Schaltzeit, oder um der Bahn eine Einfahrtszeit in Routen zu erlauben), ist es für die Sicherheit nur wichtig, dass die Zeit nicht unterschritten wird, eine Überschreitung ist für die Sicherheit unproblematisch.

7.5.4.1 Treiber

In diesem Abschnitt wird zunächst Allgemeines über das Verhalten des Treibers spezifiziert. Danach wird der Aufbau eines Treibers beschrieben. Abschließend folgt die Spezifikation von ein paar Besonderheiten von verschiedenen Treiberklassen.
Um die zwei Aufgaben, die ein Treiber hat, nämlich Verarbeiten von Anforderungen und Bekanntgabe des aktuellen Zustands, zu erfüllen, prüft jeder Treiber in regelmäßigen Abständen seine Eingabebits. Dazu wird der entsprechende Bereich des SHM kopiert, um eventuelle Konflikte, die durch gleichzeitigen Zugriff von Treiber und Steuersoftware eintstehen, zu vermeiden Liegt eine ungültige Eingabedatenkombination vor (z. B. Schaltbefehle, die für den entsprechenden Typ des Hardwareelementes nicht durchführbar sind, näheres siehe Abschnitt 7.5.3.1), wechselt der Treiber in den Fehlerzustand. In jeder Iteration werden außerdem die entsprechenden Bits für den aktuellen Zustand im SHM gesetzt. Der Treiber ergreift Maßnahmen, um den Istzustand dem Sollzustand anzupassen. Dabei sind erneute Änderungen des Sollzustandes verboten, solange der Treiber den Istzustand noch nicht angeglichen hat. Sind die entsprechenden Bits im Shared Memory gesetzt, dann bedeutet das auch, dass die Hardware, die der Treiber steuert, bereit ist, eine neue Anforderung entgegenzunehmen.

7.5.4.1.1 Aufbau des Treibers

Ein Treiber besteht aus einer Initialisierungsroutine und zwei Callback-Funktion, die vom Treiberframework aufgerufen werden. Die Namen der Funktionen müssen immer mit dem Namen der jeweiligen Treiberklasse beginnen. Beschreibung der einzelnen Funktionen:

- **Treibername_init**: Initialisierungsfunktion, die einmal beim Starten aufgerufen wird.

- **Treibername_state_change**: Diese Funktion wird aufgerufen, wenn sich die Variablen für den jeweiligen Treiber geändert haben. Sie sollte den Zustand der Hardware überprüfen, daraufhin dementsprechend mit der Hardware kommunizieren und dabei die entsprechenden Werte im Shared Memory zurückliefern.
- **Treibername_callback**: Optional kann ein Treiber auch innerhalb dieser Funktion unabhängig von Veränderungen im Shared-Memory tätig werden.

An das Ende eines Treibers gehört ein Aufruf des Makros DRV_CLASS_INIT, welches als Parameter eine Klassen-ID und den Treibernamen erhält. Durch das Makro wird eine Struktur mit Pointern auf die einzelnen Funktionen erzeugt die vom Treiberframework benötigt wird, um die einzelnen Treiber ansteuern zu können .
Beim Schreiben von Treibern ist darauf zu achten, dass kein blockierendes I/O verwendet wird. Die oben genannten Funktionen sind innerhalb kürzester Zeit abzuarbeiten und es darf nicht auf irgendwas gewartet werden. Bei Betriebssystemaufrufen ist zu überprüfen, ob sie die Echtzeiteigenschaften zu sehr negativ beeinflussen. Optimalerweise greift der Treiber direkt auf die I/O-Ports (also direkt auf die Anschlüsse, wo die Hardware angeschlossen ist) ohne Kernelbeteiligung zu.

7.5.4.1.2 Besonderheiten des Signaltreibers

Wenn mehr als ein Bit gesetzt ist, dann wechselt der Treiber in den Fehlerzustand, mit der Ausnahme, dass eine REQUEST RECEIVED-Anzeige zusätzlich zu einer anderen Anzeige gefordert werden kann. Außerdem wechselt der Treiber in den Fehlerzustand, wenn eine durch die jeweilige Signalart nicht darstellbare Anzeige gefordert wird.

7.5.4.1.3 Besonderheiten des Weichentreibers

Wenn mehrere Bits im Eingabebereich gleichzeitig gesetzt sind, oder eine für den Weichentyp unmögliche Position gefordert wird, dann wechselt der Treiber in den Fehlerzustand.

7.5.4.1.4 Besonderheiten des Sensortreibers

Findet ein Sensortreiber ein gesetztes Reset-Bit (Bit 0 im Eingabeberich, siehe Abschnitt 7.5.3.1.4) vor, dann setzt der Sensortreiber den Zustand zurück (es geht insbesondere um Bit 10 bei Route-Request-Sensoren) und löscht auch das Reset-Bit (Ausnahmsweise schreibt hier der Treiber in den Eingabebereich!).

7.5.4.2 Systemumgebung

Im folgenden wird eine Konfiguration des Systems aufgelistet, wie wir es uns vorstellen. Kriterien dafür sind: möglichst wenig verwenden, was nicht unbedingt nötig ist (aber eventuell eine zusätzliche Fehlerquelle darstellt) und Vermeidung von Komponenten, die

schon potentiell als unsicher/ausfallgefährdet gelten wie Festplatten (stattdessen soll als zuverlässiger geltender Flash-Speicher verwendet werden). Es muss dazu gesagt werden, dass es sich nur um ein Konzept handelt mit teilweise grob geschätzten Werten, welches nicht umgesetzt wurde. Als Abschluss folgt dann die Form, wie es reell gestartet wird.

7.5.4.2.1 Konzept Hardwareplattform

Beispielhaftes Embedded-System mit weitgehend Standard-PC-Hardware:
http://www.bcmcom.com/bcm_product_box3610.htm

- +256 MB Flash, +256 MB RAM (Sicherheitsreserven)
- evtl. 8255-IO-Karte [Dec]
- evtl. IOWarrior-IO-Karte [Cod]
- Unterbrechnungsfreie Stromversorgung

7.5.4.2.2 Konzept Softwareplattform

- Linux Kernel (monolithisch kompiliert, in minimaler Konfiguration)
- uclibc (minimale Variante der Glibc, einer Bibliothek mit Standard-C-Funktionen)
- alles statisch gelinkt
- evtl. zusätzliche Bibliotheken für Treiber
- busybox (stellt eine Vielzahl von Systemprogrammen in einfacher Form in einem einzelnen Programm zur Verfügung
- evtl. Fernwartungstools (ssh, telnet, ...)
- primitiver Bootloader

7.5.4.2.3 Konfiguration

Es muss einen eigenen Init-Prozess geben, der einen Prüfsummencheck für alle Dateien des Steuerinterpreters durchführt. Dabei sollen möglichst wenig Prozesse gestartet werden und das System aus einer RAM-Disk (initrd=initial ram disk) geladen werden, damit zur Laufzeit Zugriffe auf den (wahrscheinlich langsameren) persistenten Speicher zugegriffen wird.

7.5.4.2.4 Konzept Startvorgang des Systems

Bevor dieser Startvorgang durchgeführt wird, muss sichergestellt werden, dass sich keine Bahn auf den Routen befindet, dies muss nötigenfalls durch manuelles Herausleiten geschehen. Zunächst lädt der Bootloader den Kernel mit initrd-Dateisystem, der dann das root-Dateisystem auf der Ramdisk anlegt und den SI-init-Prozess. Dieser macht folgendes:

- optional: SI-init startet nicht SI relevante Prozesse
- SI-init startet Treiber
- SI-init startet eigentlichen SI-Prozess
- SI-init wartet auf Signale zum Runterfahren (USV/Ausschalter)

7.5.4.2.5 Initialisierung des Safety Controller

Der eigentliche SI-Prozess setzt die Datenstrukturen entsprechend dem angegebenen Initialzustand und ruft danach in einer Endlosschleife wie im Architektur-Kapitel beschrieben die einzelnen Komponenten auf wie in Abbildung 7.40 dargestellt. `phase_master` ist hierbei dafür zuständig das Umschalten zwischen dem RC, dem RD und dem SM zu regeln.

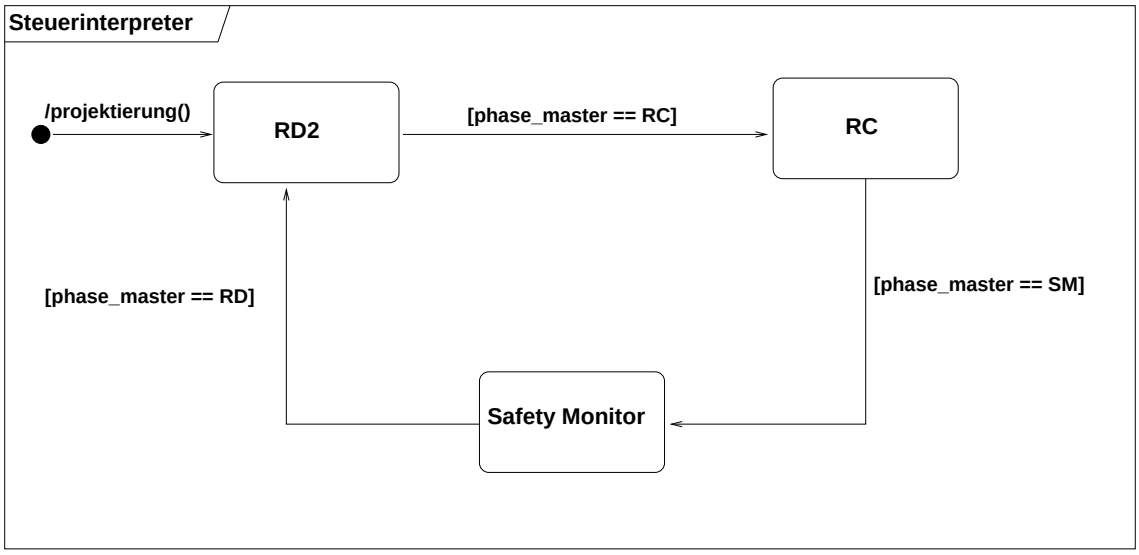


Abbildung 7.40: SI-Gesamtsystem

7.5.4.2.6 Start des Systems bestehend aus Simulator und Steuersoftware

Diese Beschreibung wurde auch so verwirklicht.

1. Projektierungsdaten generieren
2. Header für das Treiberframework generieren:

   ```
   tram-hw.sh gleisnetz.tnd
   ```

3. Treiber compilieren (dafür müssen im Simulator die Treiber bereits compiliert sein)

   ```
   cd si/src/ddk
   make CC=gcc\ -g
        DRIVER_ARCHIVE=../../../SimTracs/src/drivers/libsidriver.a
        tracs-drv
   ```

4. Im Simulator eine Simulation starten
5. tracs-drv starten
6. SI starten:

   ```
   si projdata
   ```

Am Projekttag wurden für jedes Gleisnetz eine passende `tracs-drv`-Datei bereitgehalten und das Laden der TND führte dann direkt dazu, dass das Treiberframework und der SI gestartet wurde.

7.5.4.3 Safety Monitor

In Abbildung 7.41 wird der Safety Monitor in einem Statechart dargestellt. Der Safety Monitor überprüft, ob der aktuelle Zustand und der vom RC gelieferte Sollzustand einem der in den Projektierungsdaten aufgelisteten sicheren Zuständen entsprechen. Ist das der Fall, werden die Anforderungen vom RC weitergeleitet, andernfalls werden alle Signale auf Wait gesetzt. Im folgenden werden jeweils die Eingangsdaten ,die Ausgangsdaten und die einzelnen Funktionen beschrieben.

7.5.4.3.1 Eingangsdaten

- Sollzustand geliefert vom RC (als globales int-Array, Indizes sind die SHM-IDs), Istzustand vom SHM.
- Struktur `sm_data_t` aus den Projektierungsdaten

7.5.4.3.2 Ausgangsdaten

- Sollzustand-Bereich im SHM, wo er von den Treibern dann weiterverarbeitet werden kann.

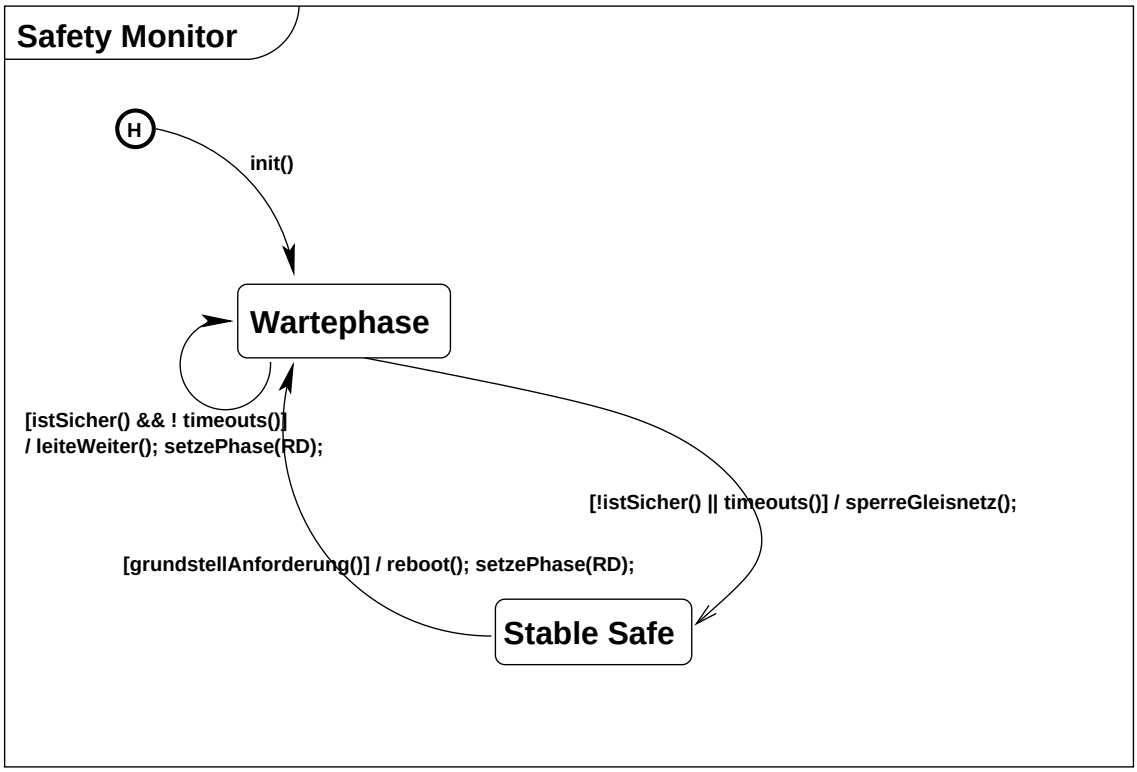


Abbildung 7.41: Safety Monitor-Transitionssystem

7.5.4.3.3 Funktionen

`istSicher()`: prüft, ob der aktuellen Zustand in der Liste sicherer Zustände enthalten ist oder ob es Sollzustandsanforderungen für Elemente gibt, deren Sollzustände nicht mit den Istzuständen übereinstimmen.
`timeouts()`: true wenn sich für mindestens ein Hardwareelement Sollzustände und Istzustände unterscheiden und der zugehörige Timer nicht läuft, sonst false
`grundstellAnforderung()`: prüft ob eine Anforderung gekommen ist, um das System in den Ausgangszustand zu setzen.

`leiteWeiter()`: leitet die Sollzustandsanforderungen vom RC/RD weiter. Unterscheidet sich der angeforderte Sollzustand vom RC/RD mit dem schon bekannten Sollzustand, wird der Timer für das jeweilige Element mit neugestartet (läuft nach der maximalen Schaltzeit ab).
`sperreGleisnetz()`: setzt alle Signale auf Wait.
Die in den Projektierungsdaten aufgelisteten sicheren Zustände sollen folgende Bedingungen gewährleisten:

1. Keine zwei Bahnen fahren in entgegengesetzte Richtungen auf demselben Gleisstück.
2. Keine zwei Bahnen fahren gleichzeitig auf denselben Sensor von verschiedenen Richtungen zu.
3. Auf jedem Gleisstück sind nicht mehr als eine vorgegebene Anzahl von Zügen gleichzeitig.
4. Bei jeder Weiche darf die Anzahl der Züge, die gleichzeitig auf den Abzweigungen fahren, ein Maximum nicht überschreiten.
5. Keine zwei Züge befinden sich gleichzeitig auf zwei unterschiedlichen Gleisstücken, die sich kreuzen.
6. Weichen haben ihren Schaltvorgang nach der maximalen Schaltzeit beendet.

Zu einem sicheren Zustand gehört auch, dass keine Signal-Anforderungen vom Route Controller vorliegen, die solche Sicherheitsverletzungen provozieren. Diese Forderung ist notwendig, um sicherheitskritsche Echtzeitanforderungen zu vermeiden.

Andreas Kemnade

7.5.4.4 Route Dispatcher

In Abbildung 7.42 wird der Route Dispatcher in einem Statechart dargestellt. Der Route Dispatcher ist dafür zuständig, zu überprüfen, welche Routen freigeschaltet werden können, ohne dass es zu Konflikten kommt. Im folgenden werden jeweils die Eingangsdaten und die Ausgangsdaten und die einzelnen Funktionen beschrieben.

7.5.4.4.1 Eingangsdaten

- Route State Table (RST)
- Route Conflict Table (RCT, aus Projektierungsdaten)

- Liste von Routen Request Sensoren (als SHM-IDs aus Projektierungsdaten)
- aktueller SHM Zustand (als Kopie)
- Queue Priority List (QPL, umsortierbare Liste von Queues, ist initial nach der ID aufwärts geordnet)
- Zuordnung Route Request zur RR-Queue (aus Projektierungsdaten)
- Bahncounter für jede Route (als int-Array, initial 0)

7.5.4.4.2 Ausgangsdaten

- Route State Table (int-Array, initial alle Routen inaktiv)
- Queue Priority List
- Bahncounter für jede Route (als int-Array, initial 0)

Die Queue Priority List liegt ebenfalls als globales int-Array vor. Die RR-Queues für die einzelnen Routen haben eine maximale Länge von 3 und sind initial leer.
Der Route Dispatcher besteht aus einer Schleife über alle Route-Request-Sensoren, die überprüft, ob eine Routenanforderung vorliegt und diese dann der jeweiligen Queue hinzufügt, und einer Schleife über alle Queues (anhand der Queue Priority List), in der dann geprüft wird, ob und welche Routen freigegeben werden können. Nun folgt eine Beschreibung der einzelnen Funktionen:

7.5.4.4.3 Funktionen

`route_state`: gibt den Shared-Memory-Zustand des RR-Sensors an.
`hat_route`: gibt an, ob es noch zu bearbeitende RR-Sensoren gibt.
`hat_queue`: gibt an, ob es noch zu bearbeitende Queues gibt.
`hat_konflikt`: gibt an, ob eine angeforderte Route nach der RCT in Konflikt zu Routen steht, die nicht im Zustand inaktiv sind
`zur_Queue_hinzufügen`: fügt den Route-Request zur in den Projektierungsdaten angegebenen Queue hinzu und setzt den jeweiligen RR-Sensor zurück.
`Rst_eintrag_aktiv`: setzt den Eintrag der jeweiligen Route auf aktiv
`setze_Rst_eintrag_angefordert`: setzt den Eintrag der jeweiligen Route auf angefordert
`ink_bahn_counter_von_router`: inkrementiert den jeweiligen Bahnzähler für eine Route
`aus_queue_löschen`: entfernt den Route Request aus der Queue

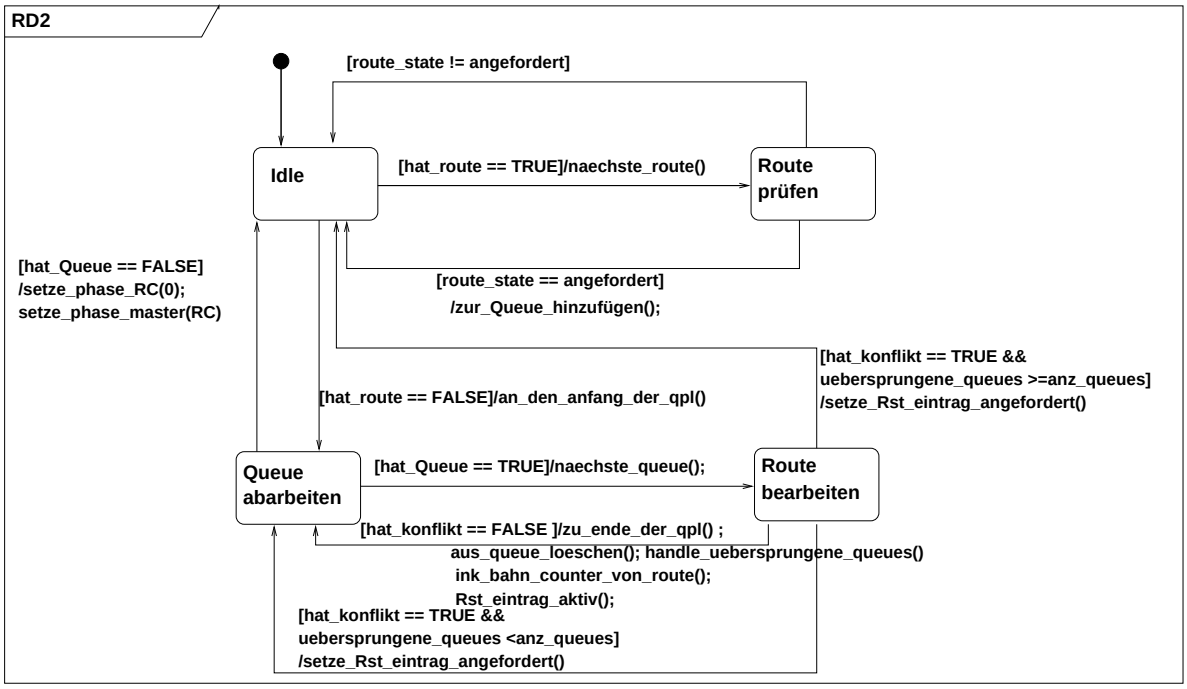


Abbildung 7.42: Route Dispatcher-Transitionssystem

`zu_ende_der_qpl`: setzt den jeweiligen Request an das Ende der Queue Priority List, dekrementiere den Schleifenzähler
`handle_uebersprungene_queues`: inkrementiert den uebersprungene_queues-Zähler, wenn es sich nicht um die erste Queue handelt, ansonsten setze den Zähler auf 0.
`naechste_queue`: gehe zur in der QPL folgenden Queue über (also inkrementiere den Schleifenzähler).
`naechste_route`: gehe zur nächsten Route über (also inkrementiere den Schleifenzähler).
`an_den_anfang_der_qpl`: gehe an den Anfang der QPL (also Schleifenzähler auf 0 setzen)

Andreas Kemnade

7.5.4.5 Route Controller

In Abbildung 7.43 wird der RC in einem Statechart dargestellt. Im folgenden werden jeweils die Eingangsdaten und die Ausgangsdaten und die einzelnen Funktionen beschrieben. Der Route Controller ist dafür zuständig auf Anforderung vom Route Dispatcher für eine Route die Weichen und Signale zu schalten, und zu überprüfen, ob sie geschaltet wurden, und die Route wieder zu sperren, wenn keine Bahnen mehr einfahren sollen.

7.5.4.5.1 Eingangsdaten

- Route State Table
- Bahnzähler
- Eingangssensorzähler
- Routenbeschreibung für die jeweilige Route (aus Projektierungsdaten), bestehend aus:
  - Anforderungsliste (Signalpositionen um Anforderungen anzuzeigen)
  - Stellliste (Weichenpositionen um die Route freigeben zu können und alle Signalpostionen, die nicht zur Freigabe führen)
  - Freigabeliste (Signalpositionen um die Route freigeben zu können und die RR-Anzeigen wieder freizugeben)
  - Einfahrts- und Ausfahrtssensor
  - Sperrliste (Signalpositionen um die Route wieder zu sperren)
  - Deaktivierungsliste (Signalpositionen, um alle Signale der Route auf rot zu schalten)
  - aktueller SHM Zustand (Kopie der SHM Schnittstelle)

7.5.4.5.2 Ausgangsdaten

- Route State Table
- neuer SHM Zustand (geänderte Kopie)
- Bahnzähler
- Eingangssensorzähler

7.5.4.5.3 Funktionen

`aktionsliste_ausfuehren()`: SHM-Zustände anhand der als Parameter angegebenen Liste setzen.
`aktionsliste_erfuellt()`: Überprüfung, ob SHM-Zustände den in der als Parameter übergebenen Liste geforderten entsprechen.
`eingangssensor_ausgeloest()`: Überprüfung, ob der Eingangssensor einer Route ausgelöst wurde
`sensor_reset()`: Rücksetzen der Sensoren (RESET-bit setzen) am Eingang und Ausgang einer Route

`nachbedingungsliste_erfuellt()`: Überprüfung, ob die Sensorzählerstände vom Eingangs- und Ausfahrtssensor gleich sind in der Nachbedinungsliste entsprechen
`sperrtimer_abgelaufen()`: Überprüfung, ob der Sperrtimer zu einer Route abgelaufen ist.
`anforderungssignal_zeigen()`: Das zu einer Route zugehörige Routenanforderungssignal zeigen.
`sperrtimer_starten()`: Starten des Sperrtimers
`setze_phase_naechster_RC()`: Übergang zum nächsten Route Controller
`hasNextRC()`: Überprüfung, ob der gerade bearbeitete Route Controller der letzte ist

Andreas Kemnade

7.5.4.6 Projektierungsdaten

In den folgen Abschnitten wird der Aufbau der Projektierungsdaten erläutert und den einzelnen Teilen des Steuerinterpreters zugeordnet.

7.5.4.6.1 Vorbemerkung

Wenn im folgenden in C-Codebeispielen die Arraynotation verwendet wird sind dies keine echten C-Arrays (sprich Pointer auf das erste Element) sondern Inline-Arrays, die Daten liegen also genau an der Stelle im Speicher wo normal der Array-Pointer stehen würde. Diese Inline-Arrays werden durch spezielle Delimiter-Einträge beendet, in der Regel ist der entsprechende ID-Eintrag negativ (Ausnahmen werden entsprechend gekennzeichnet). Wenn die Pointernotation verwendet wird so sind diese Zeiger immer als relativ zum Anfang der Projektierungsdaten zu verstehen. Sämtliche Integerwerte müssen im *Little-Endian*-Format vorliegen. Zwischen den einzelnen Abschnitten darf sich eine beliebige Menge an beliebigen Daten befinden, so lange die Projektierungsdaten nicht zu groß werden.
Folgende Daten werden vom SI für die Projektierungsdaten benötigt:

7.5.4.6.2 für die einzelnen RC-Instanzen:

Diese Liste wird nicht terminiert, da die Länge bekannt ist.

- Routen-ID (fängt bei 0 an zu zählen)
- Stellliste (benötigte Weichenpositionen um die Route freigeben zu können und alle Signalpositioonen, die nicht zur Freigabe führen)
- Freigabeliste (Signalpositionen, um die Route freizugeben)
- Sperrliste (Signalpositionen, um die Route wieder zu sperren)

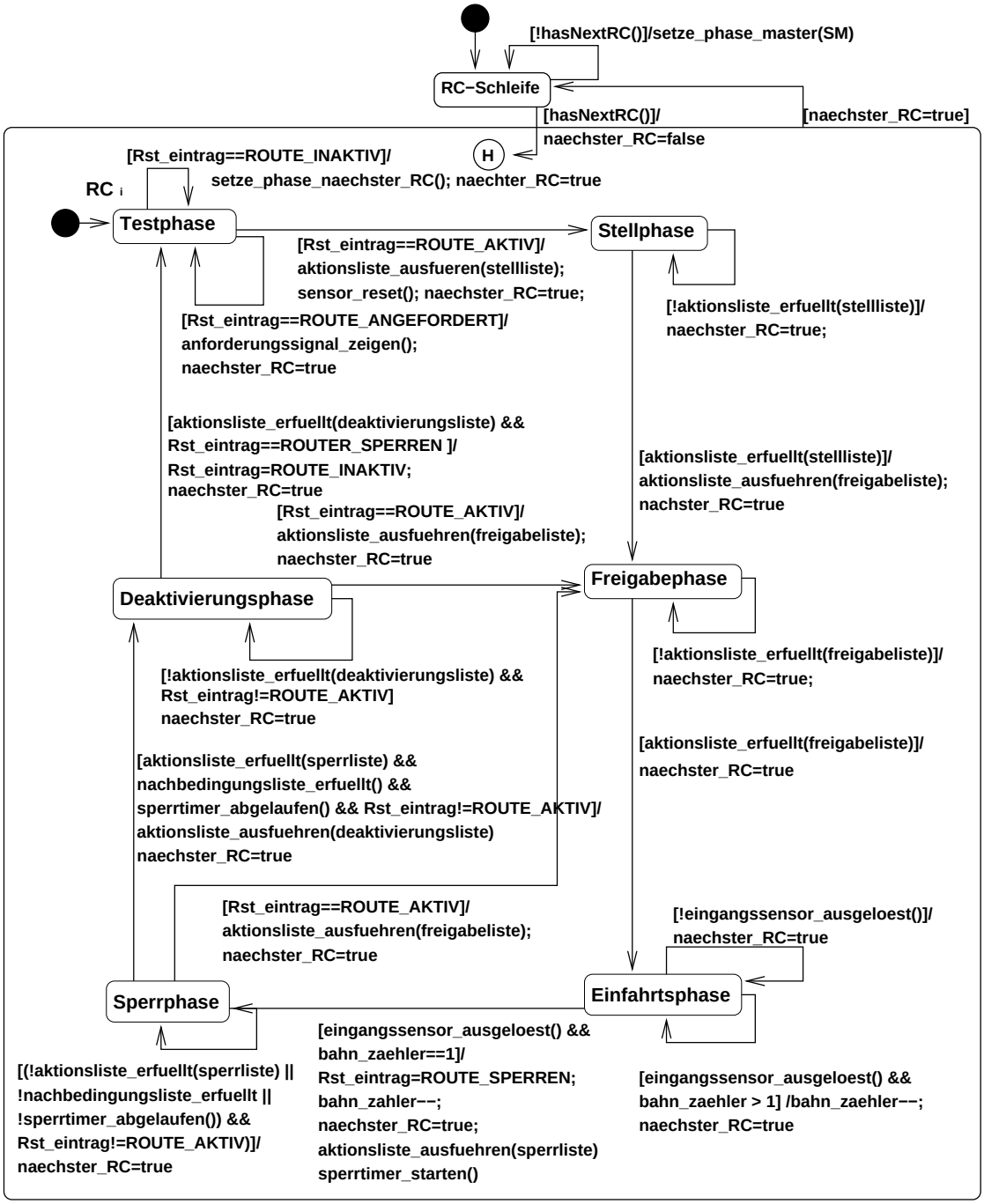


Abbildung 7.43: Route Controller-Transistionssystem

- Deaktivierungsliste (Signalpositionen, um alle Signale auf der Route auf Rot zu stellen)

- IDs von Anfang- und Endsensor

Signal- und Weichenpositionen werden dabei als SHM-Belegungsmaske gespeichert: Es werden die relevanten Bits für den Soll- und den Istzustand gesetzt und über eine Bitmaske maskiert, diese Werte werden zusammen mit der zugehörigen SHM-ID gespeichert. Als C-Struktur:

```
struct shm_value {
        int id;
        uint_16_t input;
        uint_16_t input_mask;
        uint_16_t output;
        uint_16_t output_mask;
};
```

Input bedeutet dabei: Eingaben vom SI an den Treiber
Output bedeutet: Ausgabe vom Treiber an den SI
Diese Werte sollen so gesetzt werden, dass man sie folgendermaßen verwenden kann:

```
  input=(input & (~shm_value.input_mask)) |
            (shm_value.input_mask & shm_value.input);
```

Das ist dann das, was in der RC-Spezifikation als Ausführen einer Liste bezeichnet ist. Das Testen im RC geschieht dann folgendermaßen:

```
  (output & shm_value.output_mask) ==
  (shm_value.output & shm_value.output_mask)
```

Anmerkung: Sämtliche Daten müssen gepackt gespeichert werden, *Padding* ist also nicht erlaubt.
Die Daten für einen Route Controller haben folgende Struktur:

```
struct rc_data_t {
        int route_id;
        struct shm_value_t point_pos[];
        struct shm_value_t free_sig_pos[];
        struct shm_value_t lockfirst_sig_pos[];
        struct shm_value_t lock_sig_pos[];
```

```
        struct shm_value_t req_sig_pos[];
        int entry_sens_id;
        int exit_sens_id;
};
```

Hierbei werden die „Arrays“ jeweils durch spezielle Delimitereinträge beendet.

7.5.4.6.3 für den Route Dispatcher

Zuordnung Routen zu Queues (queue_tab): Diese Tabelle dient dazu, alle Route Request Sensoren zu einer Queue zusammenzusetzen, die von annähernd demselben Ort ausgelöst werden und möglicherweise von einem einzigen physikalischen RR-Sensor ausgelöst werden. Die Indizes in diesem Array entsprechen den Routen-IDs. Die Einträge sind dann die IDs der Queues.
Route Conflict Table: Diese wird als zweidimensionales char Array implementiert. Ausserdem wird die aktuelle Anzahl der verwalteten Routen benötigt. Jedes Element innerhalb des Arrays entspricht einer Zelle in der Route Conflict Table, es werden also nur boolsche Werte gespeichert. Beispiel für eine RCT:

```
  1  2  3
1    x
2 x     x
3    x
```

```
struct rd_data_t {
        int max_routes;
        int max_queues;
        char rct[][];
        char queue_tab[];
};
```

Da die Anzahl der Routen durch `max_routes` bekannt ist werden für conflicts hier keine Delimitereinträge benötigt.
Um zu erkennen, ob zu einer Route x ein konfliktierende Route existiert, kann man die RCT dann in der folgenden Art auswerten:

```
// Auswertung für eintreffenden Route Request für Route x
for (int i = 0; i < rd_data.max_routes; i++) {
        if (route_aktiv(i) && (rd_data.rct[x][i] == 1) {
                return CONFLICT;
        }
}
return NO_CONFLICT;
```

7.5.4.6.4 für den Safety Monitor

Liste aller Sicherheitsbedingungen: Diese wird durch die Menge aller sicheren Zustände abgebildet. Diese Menge wird durch eine Liste von ODER-verknüpften Zuständen dargestellt, wobei jeder Zustand wiederum eine Liste von UND-verknüpften SHM-Zuständen (s. RC-Daten) ist, im Wesentlichen also eine Disjunktive Normalform. Ausserdem wird eine Liste von Hardware-Timeouts übergeben, und zwar als (SHM-ID, Timeout) Tupel.

```
struct sensor_value_t {
        int id1;
        int id2;
        int relation;
        int value;
};

struct sm_condition_t {
        int type;
        union {
                struct shm_value_t shm_value;
                struct sensor_value_t sensor_value;
        } condition;
};

struct hw_timeout_t {
        int id;
        time_t timeout;
};

struct sm_data_t {
        struct sm_condition_t sm_conditions[][];
        struct hw_timeout_t hw_timeouts[];
};
```

Bei `struct sensor_value_t` muss genau eins der beiden Elemente id2 oder value gesetzt sein (einen positiven Wert haben), das jeweils andere muss einen negativen Wert haben. relation gibt an, was wie verglichen werden soll. Bei relation=0 wird auf Gleichheit verglichen. Bei relation=1 wird verglichen, ob id1 kleiner ist als id2 oder value. Bei relation=2 wird verglichen, ob id1 grösser ist als id2 oder value. Bei relation=3 wird verglichen, ob id1 kleiner oder gleich ist als id2 oder value. Bei relation=4 wird verglichen, ob id1 grösser oder gleich ist als id2 oder value.

7.5.4.6.5 Gesamtformat

Die Projektierungsdaten bestehen im Gesamtformat aus den Daten für die einzelnen Teile und davon einem Block mit Offset-Daten und einem Block mit Versionsinformationen aneinandergereiht. Durch den Offset-Block kann beim Auslesen der Projektierungsdaten berechnet werden, wo die einzelnen Bereiche, die jeweils verschiedene Längen haben können, beginnen.

```
struct offsets_t {
        struct version_t *version_offset;
        struct rd_data_t *rd_data_offset;
        struct rc_data_t *rc_data_offset;
        struct sm_data_t *sm_data_offset;
};

struct version_t {
        int major;
        time_t build;
};

struct projdata {
        struct offsets_t offsets;
        struct version_t version;
        struct rd_data_t rd_data;
        struct rc_data_t rc_data[];
        struct sm_data_t sm_data;
};
```

7.5.4.7 Hardwarekonfiguration

Für die vollständige Beschreibung eines Gleisnetzes werden hardwarespezifische Informationen benötigt, um auf Ereignisse richtig reagieren zu können. So wird für jedes Hardwareelement eine eindeutige (am besten numerische) ID sowie die zugehörige Hardwareklasse benötigt. Für jede Hardwareklasse werden wiederum der Treibertyp, die Treiberklasse sowie Hardwaredaten wie z.B. Reaktionszeiten, die für den Treiber oder den Steuerinterpreter wichtig sind, angegeben. Im Projekt wurden zwei Wege bearbeitet: Zum einen eine Lösung, die aus zusätzlichen Textdateien dann die entsprechenden Header-Dateien erzeugt. Zum anderen eine Lösung, die nur die jeweiligen Simulator-Treiber verwendet. Die Vorteile der ersten Lösung konnten nicht ausgenutzt werden, da nur Simulator-Treiber zur Verfügung standen. Im folgenden werden beide Ansätze beschrieben:

7.5.4.7.1 Flexible Auswahl der Treiber

Aus der TND können die folgenden Informationen gewonnen werden: ID, Treibertyp und Mapping von ID auf Hardwareklasse. Ein paar Informationen sind jedoch nicht direkt mit in die TND aufgenommen worden, da sie vom konkreten Gleisnetz unabhängig sind. So sollten die Hardwareklassen inkl. der Hardwaredaten selbst für alle Gleisnetze gültig sein und die Zuordnung von Hardwareklassen zu Treiberklassen ist mehr Software- als Gleisnetzspezifisch (z.B. wird ein Simulator wohl andere Treiber benutzen als ein operatives System). Zur Realisierung wurde die TND um einen neuen Block erweitert, in dem die Zuordnung von Hardwareelementen zu Hardwareklassen spezifiziert wird:

```
<HWBLOCK> ::= "hardwaremap" "{" {<HWDEF>} "}"
<HWDEF>   ::= <HWCLASS> ":" <ELEMENTID> {"," <ELEMENTID} ";"
<HWCLASS> ::= "c" <IDENTIFIER>
```

Es wird also für jede Hardwareklassen eine Liste der zugehörigen Elemente angegeben. Elemente, die für den Steuerinterpreter nicht interessant sind (z.B. Markierungspunkte) haben keine Hardwareklasse und werden hier nicht aufgelistet. Ein Nachteil dieser Lösung ist, dass die Typinformationen (Treibertyp und Hardwareklasse) für jedes Element in zwei verschiedenen Listen verwaltet werden.
Die genauen Daten einer Hardwareklasse werden in folgender Form festgehalten:

```
<HWCLASSDEFS> ::= <HWCLASS> ":" <DESCRIPTION> "," <TIMEOUT> "\n"
<DESCRIPTION> ::= <LETTER-OR-DIGIT> {<DESC-CHARACTER>}
<DESC-CHARACTER> ::= <LETTER-OR-DIGIT> | " " | "_" | "-" | "+" | "."
<TIMEOUT>     ::= -1 | <INTEGER>
<INTEGER>     ::= <DIGIT> {<DIGIT>}
```

Der hier spezifizierte Timeout-Wert gibt jeweils an, in welchem Zeitraum (gemessen in Millisekunden) die Hardware auf einen Befehl des Steuerinterpreters reagieren muss, bevor ein Fehler ausgelöst wird.
Im realen Einsatz würde sich eine Art Hardware-Bibliothek mit diesen Informationen ansammeln, die man für mehrere Gleisnetze wiederverwenden könnte. Damit man diese Informationen nicht jedes Mal wieder in die TND-Daten einfügen müsste, ist es wahrscheinlich günstiger, die Daten in einer eigenen Datei zu halten. Diese Datei wird im Folgenden als „HW-Info-Datei“ bezeichnet. Man könnte die Datei mit Hilfe eines Präprozessors (z.B. den Standard C Präprozessor) in die TND einbinden, wodurch der Aufwand, diese Information zu verarbeiten, minimiert werden würde. Alternativ könnte man die Datei völlig getrennt einlesen, was aber z.B. für den Simulator einen Mehraufwand bedeutet.
Als dritte Informationsquelle dient eine weitere Textdatei, die die Zuordnungen von Treiberklassen zu Hardwareklassen enthält. Diese Zuordnung ist unabhängig vom Gleisnetz (und damit auch der TND), aber dafür wahrscheinlich abhängig vom Anwendungsszenario (z.B. Simulation, Testbetrieb, operatives System). Eine Treiberklasse ist eine Gruppe von Funktionen, die vom Steuerinterpreter für bestimmte Hardwareelemente bei bestimmten Ereignissen aufgerufen wird. Als Minimum gibt es je eine Treiberklasse für Sensoren, Signale und Weichen, in der Realität dürften es jedoch deutlich mehr sein.
Das Format für die als „Treiberinfo-Datei“ bezeichnete Datei ist wie folgt:

```
<DRV_INFO>    ::= <DESCRIPTION> "\n" <DRV_MAP>
<DRV_MAP>     ::= <DRV_CLASS> ":" <HWCLASS> {"," <HWCLASS>} "\n"
<DRV_CLASS>   ::= <LETTER> {<LETTER-OR-DIGIT>}
```

7.5.4.7.2 Nur Simulator-Treiber verwenden

Dies ist auch die Methode, wie sie für die Projekttag verwendet wurde. Dazu gibt es im Compiler-Verzeichnis ein Skript namens `tram-hw.sh`, welches die zum Compilieren des Treiberframeworks benötigten Header-Dateien aufruft. Dadurch werden den Elementen die entsprechenden Simulator-Treiber zugeordnet. Dem Skript wird der Name der TND-Datei übergeben.

Marius Mauch, Andreas Kemnade

7.5.5 Resumee der Zeit-Anforderungen

Um eine systematische Bewertung der Zeitanforderungen zu erhalten, fassen wir an dieser Stelle die an anderen Stellen im Bericht aufgetretenen Echtzeitanforderungen zusammen und geben dazu eine Bewertung ab.

- Zeit für das Durchschalten/Freigeben von Routen durch den Route Controller / Route Dispatcher: nicht sicherheitskritisch, sondern Komfortanforderung, weil es kein Sicherheitsproblem darstellt, wenn die Bahn vor einem roten Einfahrtssignal steht.

- Schalten von Signalen auf Rot durch den Safety Monitor: Da auf Sicherheitsverletzungen anhand des Sollzustands geprüft wird, werden gefährliche Anforderungen des Route Controllers gar nicht erst weitergeleitet, liegt in dem Fall dennoch kein sicherheitskritisches Zeitproblem vor. Da aber der Safety Monitor aber auch gegen selbsttätiges Schalten der Weichen aktiv sein muss, gibt es in dem Fall dennoch ein sicherheitskritsches Zeitproblem.

- Reaktion auf Bahnen, die ein rotes Signal überfährt: Hier sollte so schnell wie möglich der sichere Zustand eingenommen werden. Bei Fehlverhalten des Fahrers können aber keine Sicherheitsgarantien gegeben werden

- Erkennen von an Sensoren vorbeifahrenden Zügen: Diese Information darf auf keinen Fall verloren gehen, da Routen nicht freigegeben oder zum richtigen Zeitpunkt wieder gesperrt werden könnten.

- Reaktion auf Ablauf von Timern: Timer geben stets eine Mindestzeit (Mindestzeit, die auf das Schalten von Weichen gewartet werden soll, Mindestzeit, die eine Bahn hat, um in einer bereits gesperrten Route den ersten Sensor zu erreichen) an, wenn eine Überschreitung ist daher auch nicht sicherheitskritisch. Zur Verdeutlichung: Wenn eine Weiche die maximale Schaltzeit überschreitet, schadet es nicht, wenn ihr da vielleicht mal noch eine Sekunde zusätzliche Zeit gelassen wird.

- Hardwareansteuerung durch Treiber: Auf Grund fehlender Informationen können wir die Anforderungen nicht bewerten.

Zusammenfassend lässt sich sagen, dass sich in den meisten Fällen hieraus klar kein Bedarf für ein Hard-Realtime-System erkennen lässt, da es sich nur um Komfortanforderungen handelt, die in etwa so kritisch sind, wie die flüssige Reaktion eines normalen Programms auf Maus- oder Tastatureingaben. Das bedeutet, wenn es manchmal etwas länger dauert, dann ist das noch nicht schlimm, es sollte eben nur nicht zu oft passieren. Es ist fraglich, ob für die restlichen Fälle spezielle Echtzeitsysteme erforderlich sind, da der Steuerinterpreter sowieso nur so gut wie als einzigstes Programm auf dem Steuerrechner laufen soll. Dieser Standpunkt kann sich jedoch ändern, wenn die Treiber selbst harte Echtzeitanforderungen haben auf Grund der daran angeschlossenen Hardware. Dieser Punkt konnte jedoch auf Grund fehlender Informationen über reale Hardware nicht mehr abschließend behandelt werden.

Andreas Kemnade, Deng Zhou

7.5.6 Reflexion

In diesem Abschnitt wird zunächst dargestellt, was in den einzelnen Semestern erreicht wurde und dann erfolgt eine Zusammenstellung der wichtigsten Probleme.

7.5.6.1 Verlauf in den einzelnen Semestern

Im ersten Projektsemester erfolgte die Aneignung von einigem Grundlagenwissen und es erfolgten erste Experimente mit Shared-Memory und Transitionssystemen, teilweise wurde das auch in Assembler implementiert. Daneben wurde die Shared-Memory-Schnittstelle definiert. Im zweiten Semester wurde dann am Treiberframework gearbeitet und ein Projektierungsdatenformat entworfen, was dazu dienen konnte, ein Transitionssystem einzulesen. Es wurde spezifiziert, wie der SI diese Projektierungsdaten interpretieren soll, und es wurde eine begonnen, einen Interpreter für dieses Format zu schreiben. Das Problem dabei war aber, dass dann der Compiler extrem aufwendig sein würde und der Steuerinterpreter dann nur als ganzes getestet werden könnte. Daher wurde dann das Projektierungsdatenformat und die Spezifikation im dritten Semester neu entworfen. Erst im vierten Semester konnte die Spezifikation abgeschlossen werden und pünktlich zum Projekttag die Implementierung vorführreif erstellt werden. Fairness-Probleme mussten noch danach behoben werden.

7.5.6.2 Aufgetretene Probleme

Ein tiefer Einschnitt ergab sich nach dem zweiten Semester, da der vorherige Ansatz sich als falsch erwies. Es wurde kein Steuersystem spezifiziert, sondern nur ein Steuerinterpreter (und auch das nicht ganz vollständig), aber es wurde kaum überprüft, ob der Compiler sinnvoll entsprechende Projektierungsdaten erzeugt werden konnte, so dass sich ein Steuersystem ergibt und auch nicht, ob das Ganze dann verifiziert werden kann. Dadurch musste das meiste ausserhalb des Treiberframeworks neu spezifiziert und implementiert werden. Gerade die Spezifikation erwies sich als großer Brocken. Das Problem bestand zum einen darin, dass es mehrere Ansätze der Spezifikation gegeben hat, über die nur schwer Einigung erzielt werden konnte. Dadurch entstand trotz der Maßnahmen um die Dokumentation zu strukturieren eine unübersichtliche Dokumentstruktur, was das Erlangen von Feedback im Plenum erschwerte. Ein weiteres Problem, war dass Diagramme und Text lange nicht übereinstimmten. Das wurde unter anderem dadurch verursacht, dass es schwer fiel zur Textbeschreibung sich Zustände und Transitionen vorzustellen, wenn es sich um eine einfache Schleife handelte. Als das gelungen ist, war es dann nicht mehr schwer, zu den Funktionsnamen Text zu schreiben. Die Implementierung gestaltete sich dann nicht mehr als besonders schwierig.

Andreas Kemnade

7.6 Model Checker

7.6.1 Überblick

Model Checking ist eine relativ neue Verifikationsmethode in der Informatik, bei der in den letzten Jahren große Fortschritte erzielt werden konnten und die dadurch auch für die Praxis interessant geworden ist.
Im Unterschied zu anderen Verifikationstechniken handelt es sich beim Model Checking um einen automatischen Ansatz. Seine Einsatzmöglichkeiten sind daher auf Systeme mit einer endlichen Anzahl von Zuständen beschränkt.

7.6.1.1 Prinzip des Model Checkings

Mit Model Checking lässt sich überprüfen, ob eine bestimmte Eigenschaft in einem bestimmten Transitionssystem erfüllt ist. Die Verifikation mit Model Checking umfasst drei Phasen:

- Modell

  Es wird durch Abstraktion ein formales Modell des Systems erstellt. Das Modell muss in Form eines endlichen Transitionssystems beschrieben werden.

- Formeln

  Es werden durch Formalisierung Eigenschaften aus einer Spezifikation abgeleitet. Diese Eigenschaften können in einer temporalen Logik spezifiziert werden. Mit temporalen Logiken lassen sich zeitliche Zusammenhänge ausdrücken.

- Model Checker

  Es existieren verschiedene Tools, die die Verifikation mit Model Checking Techniken unterstützen. Das in der verwendeten Sprache des Model Checkers modellierte System und die in dem entsprechenden Formalsimus spezifizierten Eigenschaften bilden die Eingabe für den Model Checker, der automatisch prüft, ob die jeweiligen Eigenschaften erfüllt sind. Ist eine Eigenschaft verletzt, wird häufig ein Gegenbeispiel erzeugt. Durch die Rückgabe eines expliziten Gegenbeispiels können mögliche Fehlerkorrekturen vereinfacht werden.

7.6.1.2 Gliederung

Zunächst wird im Abschnitt 7.6.2 auf der nächsten Seite das Konzept zur Verifikation der Projektierungsdaten erläutert. Dann wird im Abschnitt 7.6.3 auf Seite 207 der verwendete Model Checker NuSMV vorgestellt. Wie das System modelliert wird und wie die Sicherheitsbedingungen spezifiziert werden, wird im Abschnitt 7.6.4 auf Seite 213 beschrieben. Der Abschnitt 7.6.5 auf Seite 233 beschäftigt sich damit, wie die Eingabedatei

für den Model Checker automatisch erzeugt werden kann. Wie die in den Verschlusstabellen gemachten Angaben überprüft werden können, wird im Abschnitt 7.6.6 auf Seite 239 beschrieben. Schließlich folgt im Abschnitt 7.6.7 auf Seite 252 noch eine abschließende Reflexion.

Ruben Rothaupt, Taffou Happi

7.6.2 Konzept zur Verifikation der Projektierungsdaten

In diesem Abschnitt wird beschrieben, wie sich nachweisen lässt, dass die Projektierungsdaten die notwendigen Sicherheitsanforderungen erfüllen.
Um zu überprüfen, ob alle Sicherheitsbedingungen erfüllt sind, wird Model Checking verwendet. Dazu muss ein formales Modell des Systems erstellt werden und die Sicherheitsbedingungen müssen spezifiziert werden.

7.6.2.1 Sicherheitsbedingungen

Die folgenden Sicherheitsbedingungen müssen in jedem Gleisnetz erfüllt sein. Dabei kann man zwischen Kollisionen und Entgleisungen unterscheiden.

- Kollisionen
  - Keine Bahnen dürfen aus unterschiedlichen Richtungen auf dieselbe Kreuzung zufahren, so dass es zu einem Flankenzusammenstoß kommen kann.
  - Keine Bahnen dürfen aus unterschiedlichen Richtungen auf dieselbe Weiche zufahren, so dass es zu einem Flankenzusammenstoß kommen kann.
  - Keine Bahnen dürfen in entgegengesetzter Fahrtrichtung aufeinander zufahren, so dass es zu einem Frontalzusammenstoß kommen kann.
- Entgleisungen
  - Keine Weichen dürfen umgeschaltet werden, während sie befahren werden.
  - Wenn das Auffahren nicht möglich ist, dürfen keine Bahnen auf die Weiche auffahren.

Aus der Gleisnetzbeschreibung lassen sich die Hazards bestimmen, die in dem jeweiligen Gleisnetz auftreten können. Der Controller muss die Signale und Weichen so schalten, dass keine Sicherheitsbedingung verletzt wird.

7.6.2.2 Systemmodell

Es wird ein formales Modell des Systems in Form eines Transitionssystems erstellt. Dieses Modell besteht aus einem physikalischen Modell des betrachteten Gleisnetzes und einem Modell des Controllers.

7.6.2.2.1 Physikalisches Modell (Domain of Control) Das physikalische Modell stellt eine Abstraktion des betrachteten Gleisnetzes dar. Es werden nur die physikalischen Komponenten des Gleisnetzes (Sensoren, Weichen, Signale) ohne Steuerungsaufgabe modelliert.
Anhand der Sensor-, Signal- und Weichenzustände kann man bestimmen, wann ein Sensor von einer Bahn erreichbar ist.
Ein spezieller Teil des physikalischen Modells sind die Sicherheitsbedingungen, die ebenfalls vom Aufbau des jeweiligen Gleisnetzes abhängen.

7.6.2.2.2 Controller Modell (Controller) Im Controller Modell wird das Verhalten des Steuerinterpreters und somit die eigentliche Steuerungsaufgabe modelliert.
Die Abbildung 7.44 zeigt physikalisches Modell, Controller Modell und die Beziehung zwischen beiden Modellen. Das physikalische Modell liefert die Informationen, in welchen Zuständen sich die Sensoren, Signale und Weichen befinden. Das Modell des Controllers liefert die eigentlichen Steuerungsinformationen. Abhängig von den Informationen aus dem physikalischen Modell werden bestimmte Signal- und Weichenzustände angefordert. Durch diese Anforderungen muss vermieden werden, dass ein Zustand erreicht werden kann, in dem eine Sicherheitsbedingung nicht erfüllt ist.

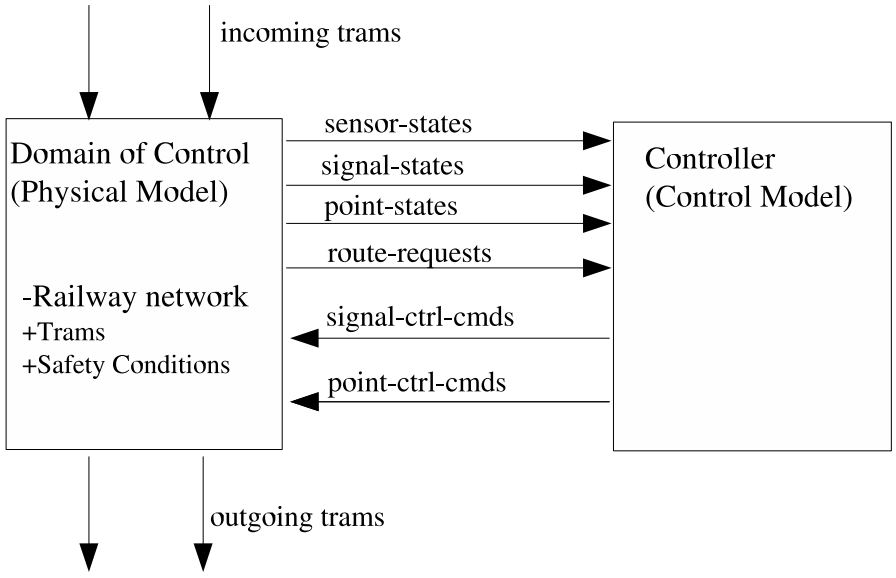


Abbildung 7.44: Systemmodell ([PGHD04])

Für den Safety Monitor wird eine Menge von Zuständen generiert, in denen sich das System befinden muss, damit keine Hazards auftreten. Ob diese Zustände gelten, wird ebenfalls überprüft. Im Gegensatz zu den oberen Sicherheitsbedingungen werden diese Zustände anhand der in den Verschlusstabellen gemachten Angaben erzeugt und sind daher von der Richtigkeit dieser Angaben abhängig.

7.6.2.3 Entwurfsprozess

Das Modell muss in der Eingabesprache des verwendeten Model Checkers NuSMV erstellt werden. Die Abbildung 7.45 zeigt, wie die Eingabedatei für den Model Checker automatisch erzeugt werden kann.
Die Informationen, welche zum Erstellen des physikalischen Modells benötigt werden, erhält man aus der Gleisnetzbeschreibung in der TND.
Der Compiler erzeugt aus Routen- und Hardwarebeschreibung die Projektierungsdaten für das jeweilige Gleisnetz. Die Informationen, welche zum Erstellen des Controller Modells benötigt werden, erhält man aus den Projektierungsdaten und der Spezifikation des Systems.

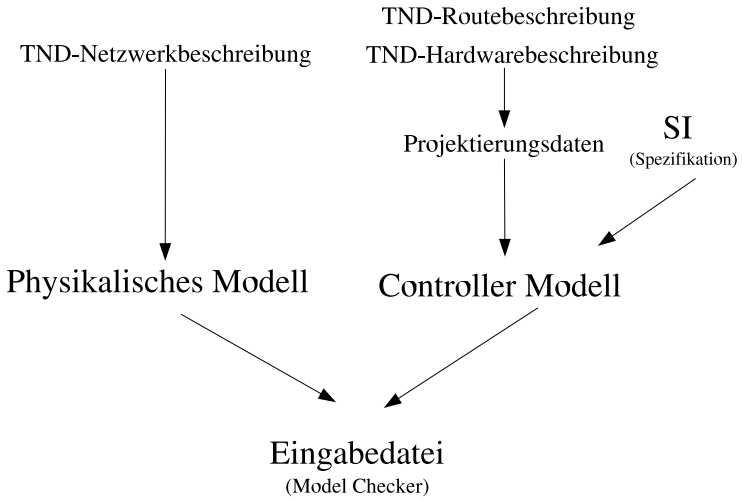


Abbildung 7.45: Model Checking Entwurfsprozess

Der Model Checker prüft dann, ob alle spezifizierten Eigenschaften erfüllt sind.

7.6.3 NuSMV: New Symbolic Model Verification

Als Model Checker wird NuSMV eingesetzt. NuSMV verwendet zur Erstellung seiner Eingabedatei eine eigene Eingabesprache, deren Hauptfunktionalitäten vorgestellt werden. Zudem wird anhand eines Beispiels beschrieben, wie ein modelliertes System überprüft werden kann. Eine ausführliche Beschreibung von NuSMV ist in [CCO+] zu finden.

7.6.3.1 Eingabesprache

Ein NuSMV-Programm besteht aus einer beliebigen Anzahl von Modulen. Das nach dem Schlüsselwort *MODULE* folgende *atom* ist der Name des Moduls. In Klammern können einem Modul Parameter übergeben werden. Das Modul mit dem Namen *main* ist ein spezielles Modul, das keine Parameter haben darf. Es dient als Evaluator für den NuSMV-Interpreter. Jedes ausführbare NuSMV-Programm muss genau ein *main*-Modul enthalten.
Folgende Bestandteile eines Moduls werden später benötigt, um das System zu modellieren:

```
"MODULE" atom [ "(" atom "," atom "," ... atom ")" ]
 [ var_declaration ]
 [ assign_declaration ]
 [ define_declaration ]
 [ spec_declaration ]
 [ checkinvar_declaration ]
 ...
```

In der *var_declaration*-Umgebung werden Variablen erzeugt, die Transitionsbedingungen der Variablen werden in der *assign_declaration*-Umgebung definiert und in der *define_declaration*-Umgebung werden zusätzliche Definitionen ausgedrückt. Schließlich werden in der *spec_declaration*-Umgebung und der *checkinvar_declaration*-Umgebung die zu prüfenden Eigenschaften als CTL-Formeln bzw. Invarianten spezifiziert.

7.6.3.1.1 var_declaration Die für die Modellierung des Systems benötigten Variablen werden in der *var_declaration*-Umgebung erstellt, die mit dem Schlüsselwort *VAR* eingeführt wird.
Variablen werden auf folgende Arten definiert:

- var : boolean;

  Die Variable `var` ist vom Typ `boolean` und kann die Werte 0 und 1 annehmen, die false bzw. true repräsentieren.

- var : 0..9;

  Die Variable `var` kann Werte innerhalb eines definierten Zahlenbereiches annehmen (hier von 0 bis 9).

- var : {value_1, ..., value_n};

  Es wird eine beliebige Anzahl von Begriffen definiert. Die Variable `var` nimmt jeweils einen dieser Begriffe an.

- var : module(param_1, ..., param_n);

  Eine Variable kann auch eine Instanz eines Moduls sein. Dazu wird die jeweilige Anzahl von Parametern übergeben. Hier ist die Variable `var` eine Instanz des Moduls `module`. Sie kann nun auf alle Variablen des betreffenden Moduls zugreifen. Wenn das Modul `module` z.B. eine Variable `var1` enthält, erfolgt der Zugriff mit `var.var1`, wodurch eine hierarchische Struktur zwischen den einzelnen Modulen entsteht.

7.6.3.1.2 assign_declaration Nachdem die Variablen erstellt sind, müssen die Transitionsbedingungen für die Variablen definiert werden.
Dies geschieht in der *assign_declaration*-Umgebung, die mit dem Schlüsselwort *ASSIGN* eingeführt wird.

- Startzustand

  Mit dem Schlüsselwort *init* wird einer Variable bei der Systeminitialisierung ein Anfangswert zugeordnet. Dieser Wert muss dabei vom zuvor definierten Typ sein.

  ```
  init(var) := value;
  ```

- Transitionsbedingungen

  Mit dem Schlüsselwort *next* wird einer Variable der jeweilige Folgezustand zugeordnet. Abhängig davon, welche Bedingung erfüllt ist, wird der betreffende Folgezustand angenommen.

  ```
  next(var) := case
                condition_1 : value_1;
                ......................
                condition_n : value_n;
               esac;
  ```

7.6.3.1.3 define_declaration In der *define_declaration*-Umgebung lassen sich Definitionen erstellen. Diese Umgebung wird mit dem Schlüsselwort *DEFINE* eingeführt.

- `definition := value;`
- ```
  definition := case
                  condition_1 : value_1;
                  ......................
                  condition_n : value_n;
                esac;
  ```

7.6.3.1.4 spec_declaration In dieser Umgebung werden Eigenschaften in der temporalen Logik CTL (Computation Tree Logic) spezifiziert. CTL ist eine Logik mit verzweigter Zeit, d.h. es können für einen Zustand mehrere Folgezustände existieren. Das Schlüsselwort *SPEC* drückt aus, dass eine Eigenschaft folgt, die in CTL formuliert ist. Eine CTL-Formel ist:

- eine atomare Formel p, true oder false
- wenn p und q CTL-Formeln sind, dann auch:

  ```
  !p, p & q, p | p, p -> q,
  AX, AF, AG, A[p U q],
  EX, EF, EG, E[p U q]
  ```

Die temporalen Verknüpfungen sind jeweils aus einem Pfadquantor (gibt an, für welche Pfade eines Transitionssystems eine Eigenschaft gilt) und einem temporalen Operator zusammengesetzt. Es existieren folgende Pfadquantoren bzw. temporale Operatoren:

- Pfadquantoren
  - **A** (**A**lways) „entlang aller Pfade “
  - **E** (**E**xists) „entlang eines Pfades “
- temporale Operatoren
  - **X** (ne**X**t) „nächster Zustand “
  - **F** (**F**uture) „in einem zukünftigen Zustand “
  - **G** (**G**lobally) „in allen zukünftigen Zuständen “
  - **U** (**U**ntil) „bis “

Folgende CTL-Formeln haben z.B. folgende Bedeutungen:
  ```

- AG p

  Entlang aller Pfade im Transitionssystem ist p in allen zukünftigen Zuständen erfüllt.

- EG p

  Es existiert ein Pfad im Transitionssystem, in dem p in allen zukünftigen Zuständen erfüllt ist.

- AF p

  Entlang alle Pfade im Transitionssystem wird irgendwann ein Zustand erreicht, in dem p erfüllt ist.

- A[p U q]

  Entlang aller Pfade im Transitionssystems ist p solange erfüllt, bis q erfüllt ist.

7.6.3.1.5 checkinvar_declaration In der *checkinvar_declaration*-Umgebung werden Invarianten spezifiziert. Dies sind Bedingungen, die in allen Zuständen des Systems erfüllt sein müssen. Das Schlüsselwort *INVARSPEC* drückt aus, dass eine Invariante folgt.

7.6.3.2 Beispiel

Es soll nun ein einfaches Beispiel angegeben werden. Das Beispiel dient dazu, das Grundprinzip von NuSMV zu verdeutlichen, bevor im nächsten Abschnitt die Modellierung das eigentlichen Systems beschrieben wird.
Die Abbildung 7.46 zeigt eine einfache Kreuzung. Vor der Kreuzung befindet sich jeweils ein Signal und hinter der Kreuzung ein Sensor.

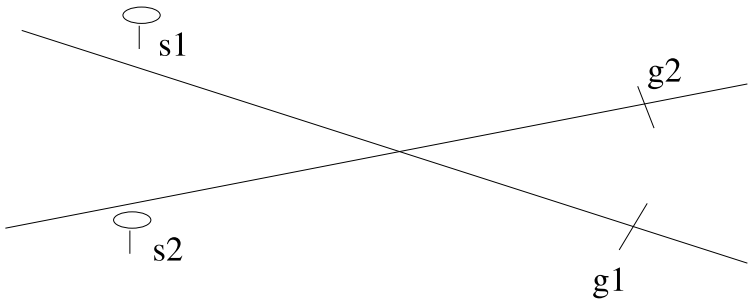


Abbildung 7.46: Einfaches Gleisnetz

Es wird nun ein NuSMV-Programm erstellt, das die Sicherheit der Kreuzung nachweist. So darf die Kreuzung niemals von verschiedenen Richtungen gleichzeitig erreichbar sein und die Kreuzung muss aus beiden Richtungen irgendwann zu erreichen sein.
Für das Beispielgleisnetz werden Variablen für die Signale `s1` und `s2` erstellt. Die beiden Signale sollen die Werte `STOP` und `GO` annehmen können. Ein Signal soll in den Zustand `GO` wechseln, wenn es zuvor im Zustand `STOP` war und in den Zustand `STOP`, wenn es zuvor im Zustand `GO` war. Außerdem wird definiert, dass ein Sensor erreichbar ist, wenn sich das zugehörige Signal im Zustand `GO` befindet und nicht erreichbar ist, wenn sich das zugehörige Signal im Zustand `STOP` befindet.

```
MODULE main

VAR
s1 : {GO, STOP};
s2 : {GO, STOP};

ASSIGN
init(s1) := GO;
init(s2) := STOP;

next(s1) := case
             s1 = STOP : GO;
             s1 = GO   : STOP;
            esac;

next(s2) := case
             s2 = STOP : GO;
             s2 = GO   : STOP;
            esac;

DEFINE
g1_reachable := case
                 s1 = GO   : 1;
                 s1 = STOP : 0;
                esac;

g2_reachable := case
                 s2 = GO   : 1;
                 s2 = STOP : 0;
                esac;

safe := !((s1 = GO) & (s2 = GO));

INVARSPEC safe
INVARSPEC !(g1_reachable & g2_reachable)
SPEC AG safe
SPEC AG !(g1_reachable & g2_reachable)
SPEC AF g1_reachable
```

```
SPEC AF g2_reachable
```

Die Bedingung `safe` ist wahr, wenn die Signale `s1` und `s2` nicht gleichzeitig auf `GO` gesetzt sind. Mit der folgenden Invariante wird überprüft, ob dies in allen Zuständen des Systems gilt.

```
INVARSPEC safe
```

Die Eigenschaft kann alternativ auch folgendermaßen spezifiziert werden:

```
SPEC AG safe
```

Die weiteren Eigenschaften überprüfen, ob die beiden Sensoren niemals gleichzeitig erreichbar sind und ob beide Sensoren in jedem Fall irgendwann erreicht werden können.

7.6.3.2.1 Ausführen von NuSMV Die definierten Eigenschaften können nun automatisch überprüft werden. Dies geschieht durch Eingabe des folgenden Befehls, wobei mc_bsp.smv der Name des Beispielprogramms ist:

```
NuSMV mc_bsp.smv
```

In der Ausgabe kann man erkennen, dass alle Eigenschaften mit true ausgewertet werden.

```
-- specification AG safe is true
-- specification AG (!(g1_reachable & g2_reachable)) is true
-- specification AF g1_reachable is true
-- specification AF g2_reachable is true
-- invariant safe  is true
-- invariant !(g1_reachable & g2_reachable)  is true
```

7.6.3.2.2 Simulation Zudem kann man eine Simulation des modellierten Systems ausführen. Dadurch lässt sich beobachten, wie sich das Modell in bestimmten Situationen verhält.
Mit dem Kommando *NuSMV -int mc_bsp.smv* kann man in den interaktiven Modus wechseln.
Danach lässt sich mit dem Kommando *go* das Model einlesen und mit dem Kommando *pick_state -r* ein zufälliger Startzustand auswählen. Die Simulation wird mit dem Kommando *simulate* gestartet, wobei angegeben werden muss, wie lang der zu erzeugende Pfad sein soll und auf welche Weise er erzeugt werden soll. Mit der Option *-r* wird z.B. ein zufälliger Pfad erzeugt.
Der aktuelle Zustand der Simulation kann mit dem Kommando *print_current_state -v* ausgegeben werden und der erzeugte Pfad mit dem Kommando *show_traces -v*.
Um eine neue Simulation zu starten, lässt sich mit dem Kommando *goto_state* ein existierender Zustand als Anfangszustand der neuen Simulation auswählen.

```
NuSMV > go
NuSMV > pick_state -r
NuSMV > print_current_state -v
Current state is 1.1
safe = 1
g2_reachable = 0
g1_reachable = 1
s1 = GO
s2 = STOP
NuSMV > simulate -r 2
********  Simulation Starting From State  1.1  ********
NuSMV > show_traces -v
################### Trace number: 1 ###################
-> State 1.1 <-
    safe = 1
    g2_reachable = 0
    g1_reachable = 1
    s1 = GO
    s2 = STOP
-> State 1.2 <-
    safe = 1
    g2_reachable = 1
    g1_reachable = 0
    s1 = STOP
    s2 = GO
-> State 1.3 <-
    safe = 1
    g2_reachable = 0
    g1_reachable = 1
    s1 = GO
    s2 = STOP
NuSMV > goto_state 1.3
The starting state for new trace is:
-> State 2.3 <-
    safe = 1
    g2_reachable = 0
    g1_reachable = 1
    s1 = GO
    s2 = STOP
```

Eine ausführliche Beschreibung ist ebenfalls in [CCO+] zu finden.

Ruben Rothaupt, Taffou Happi

7.6.4 Überprüfung des Systemmodells

In diesem Abschnitt soll das Modell des Systems dargestellt werden.

Dazu wird das Verhalten der Sensoren, der Weichen und der Signale modelliert. Zudem wird modelliert, wann eine Route welchen Zustand annimmt und wann eine Route angefordert werden kann. Schließlich folgt noch die Spezifikation der Sicherheitsbedingungen, die in dem jeweiligen Gleisnetz gelten müssen.

7.6.4.1 Beispiel

Das Modell wird anhand eines Beispiels erläutert. Als Beispielgleisnetz wird die Teilstrecke 4 der Teststrecke verwendet, die in Abbildung 7.47 zu sehen ist.
Aus der Gleisnetzbeschreibung kann das physikalische Modell erstellt werden. Unten werden die Verschlusstabellen für das Beispiel angegeben. Aus den dort enthaltenen Informationen werden die Projektierungsdaten erzeugt, die zum Erstellen des Controller Modells benötigt werden.

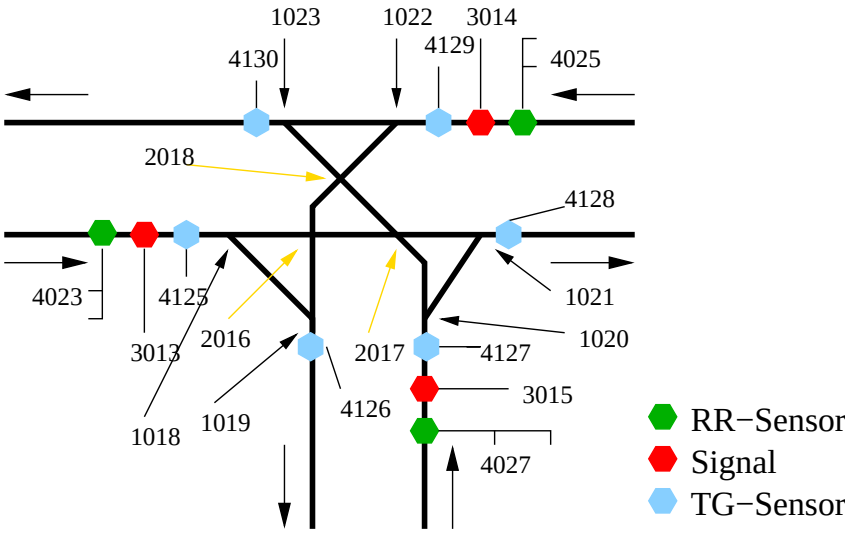


Abbildung 7.47: Strecke 4

Im Route Definitions Table werden sechs Routen definiert. Für jede Route werden die Sensoren angegeben, die auf der Route liegen. Zudem wird jeder Route ein Route Request Sensor zugeordnet. In dem Beispielgleisnetz gibt es insgesamt drei Route Request Sensoren. Die Namen der Routen und der Route Request Sensoren werden hier aus den Projektierungsdaten und nicht aus den eigentlichen Verschlusstabellen übernommen, damit sie in der Modellierung wiedererkannt werden können.

Route Definition Table

Route	Route Sensor Sequence	Route Request Sensor
r0	(g4125, g4128)	0
r1	(g4125, g4126)	0
r2	(g4129, g4130)	1
r3	(g4129, g4126)	1
r4	(g4127, g4130)	2
r5	(g4127, g4128)	2

Im Point Position Table wird definiert, welche Weichenstellungen für die jeweilige Route benötigt werden.

Point Position Table

Route	w1018	w1020	w1022
r0	straight	-	-
r1	right	-	-
r2	-	-	straight
r3	-	-	left
r4	-	straight	-
r5	-	right	-

Im Signal Setting Table wird definiert, welche Signalstellungen für die jeweilige Route benötigt werden.

Signal Setting Table

Route	Signal	Setting
r0	s3013	straight
r1	s3013	right
r2	s3014	straight
r3	s3014	left
r4	s3015	straight
r5	s3015	right

Im Route Conflict Table werden die Konflikte zwischen den Routen definiert.

Route Conflict Table

Route	r0	r1	r2	r3	r4	r5
r0		**X**		X	X	X
r1	**X**			X		
r2				**X**	X	
r3	X	X	**X**		X	
r4	X		X	X		**X**
r5	X				**X**	

7.6.4.2 Weichenverhalten

Zunächst wird die Modellierung des Weichenverhaltens dargestellt.

7.6.4.2.1 Allgemeines Weichenverhalten In dem Modul `point` wird das Verhalten beschrieben, das für alle Weichen gleich ist. Demzufolge ist dieser Teil gleisnetzunabhängig.
Als Parameter werden drei Bedingungen übergeben, die angeben, ob eine Anforderung existiert, die entsprechende Weiche geradeaus, nach links oder nach rechts zu stellen. Dabei sind nur Anforderungen möglich, die mit dem jeweiligen Weichentyp vereinbar sind. Eine SL-Weiche kann also beispielsweise nicht nach rechts gestellt werden.
Die Variable `state` repräsentiert den aktuellen Zustand der Weiche und kann die Werte `STRAIGHT`, `RIGHT`, `LEFT` und `UNDEFINED` annehmen. `UNDEFINED` drückt aus, dass die jeweilige Weiche noch nicht gestellt wurde. Dies ist der Initialzustand einer Weiche. Ist nun eine Bedingung zum Umschalten der Weiche erfüllt, nimmt diese den entsprechenden Zustand an. Ansonsten behält die Weiche ihren vorherigen Zustand bei.

```
MODULE point(straight_req, left_req, right_req)

VAR
state : {STRAIGHT, RIGHT, LEFT, UNDEFINED};

ASSIGN
init(state) := UNDEFINED;

next(state) := case
      straight_req : STRAIGHT;
      left_req : LEFT;
      right_req : RIGHT;
      1 : state;
      esac;
```

7.6.4.2.2 Initialisierung der Variablen Für alle in einem Gleisnetz enthaltenen aktiven Weichen werden Variablen erstellt. Dazu wird jeweils das Modul `point` aufgerufen und als Parameter werden die Bedingungen übergeben, die angeben, wann die Weiche den jeweiligen Zustand annehmen soll. Damit nur Anforderungen möglich sind, die zu dem Weichentyp passen, ist `none` als Bedingung definiert, die niemals erfüllt sein kann.

```
w1022 : point(w1022_straight, w1022_left, none);
w1018 : point(w1018_straight, none, w1018_right);
w1020 : point(w1020_straight, none, w1020_right);
```

7.6.4.2.3 Bedingungen zum Stellen der Weichen Für jede Weiche existieren Bedingungen, wann diese in eine bestimmte Richtung gestellt werden soll.
Wenn sich eine Route im Zustand `ACTIVE` befindet, die die entsprechende Weichenstellung benötigt, ist die Bedingung zum Schalten der Weiche erfüllt.
In dem Beispiel soll die Weiche `w1022` geradeaus gestellt werden, wenn die Route `r2` im Zustand `ACTIVE` ist und nach links, wenn die Route `r3` im Zustand `ACTIVE` ist.

```
w1022_straight := case
      r2.state = ACTIVE : 1;
      1 : 0;
      esac;

w1022_left := case
      r3.state = ACTIVE : 1;
      1 : 0;
      esac;

w1018_straight := case
      r0.state = ACTIVE : 1;
      1 : 0;
      esac;

w1018_right := case
      r1.state = ACTIVE : 1;
      1 : 0;
      esac;

w1020_straight := case
      r4.state = ACTIVE : 1;
      1 : 0;
      esac;

w1020_right := case
      r5.state = ACTIVE : 1;
      1 : 0;
      esac;
```

7.6.4.3 Signalverhalten

Im Folgenden wird die Modellierung des Signalverhaltens beschrieben.

7.6.4.3.1 Allgemeines Signalverhalten In dem Modul `signal` wird das Verhalten beschrieben, das für alle Signale gleich ist.
Als Parameter werden vier Bedingungen übergeben, die angeben, ob eine Anforderung existiert, das entsprechende Signal auf STOP, geradeaus, nach links oder nach rechts zu schalten.

Die Variable `state` repräsentiert den aktuellen Zustand des Signals und kann die Werte `STOP`, `STRAIGHT`, `RIGHT` und `LEFT` annehmen. Der Initialzustand eines Signals ist `STOP`. Ist nun eine Bedingungen zum Umschalten des Signals erfüllt, nimmt dieses den entsprechenden Zustand an. Ansonsten behält das Signal seinen vorherigen Zustand bei.

```
MODULE signal(stop_req, straight_req, left_req, right_req)

VAR
state : {STOP, STRAIGHT, RIGHT, LEFT};

ASSIGN
init(state) := STOP;

next(state) := case
      stop_req : STOP;
      straight_req : STRAIGHT;
      left_req : LEFT;
      right_req : RIGHT;
      1 : state;
      esac;
```

7.6.4.3.2 Initialisierung der Variablen Für alle in einem Gleisnetz enthaltenen Signale werden Variablen erstellt. Dazu wird jeweils das Modul `signal` aufgerufen und als Parameter werden die Bedingungen übergeben, die angeben, wann das Signal den jeweiligen Zustand annehmen soll. Auch hier ist nur das Anfordern von Signalstellungen möglich, die der jeweilige Signaltyp auch erlaubt.

```
s3014 : signal(s3014_stop, s3014_straight, s3014_left, none);
s3013 : signal(s3013_stop, s3013_straight, none, s3013_right);
s3015 : signal(s3015_stop, s3015_straight, none, s3015_right);
```

7.6.4.3.3 Bedingungen zum Stellen der Signale Für jedes Signal existieren Bedingungen, wann es einen bestimmten Zustand annehmen soll.
Wenn eine Route, die die entsprechende Signalstellung benötigt, im Zustand `ACTIVE` ist und alle Weichen, die zu dieser Route gehören, bereits richtig gestellt wurden, ist die Bedingung zum Schalten des Signals erfüllt.
Es müssen erst alle Weichen so gestellt sein, wie es für die jeweilige Route definiert wurde, bevor durchs Stellen der Signale die Route freigegeben werden darf.
In dem Beispiel soll das Signal `s3014` geradeaus gestellt werden, wenn die Route `r2` im Zustand `ACTIVE` und die Weiche `w1022` geradeaus gestellt ist und nach links, wenn die Route `r3` im Zustand `ACTIVE` und die Weiche `w1022` nach links gestellt ist.
Wenn eine Route, zu der das Signal gehört, wieder gesperrt werden soll, muss das betreffende Signal auf `STOP` gestellt werden.

So soll das Signal `s3014` in dem Beispiel auf `STOP` gestellt werden, wenn entweder die Route `r2` oder die Route `r3` wieder gesperrt werden soll. Die entsprechenden Bedingungen, wann eine Route gesperrt werden soll, werden weiter unten angegeben.

```
s3014_stop := case
      r2_lock_cond : 1;
      r3_lock_cond : 1;
      1 : 0;
      esac;

s3014_straight := case
      r2.state = ACTIVE & w1022.state = STRAIGHT : 1;
      1 : 0;
      esac;

s3014_left := case
      r3.state = ACTIVE & w1022.state = LEFT : 1;
      1 : 0;
      esac;

s3013_stop := case
      r0_lock_cond : 1;
      r1_lock_cond : 1;
      1 : 0;
      esac;

s3013_straight := case
      r0.state = ACTIVE & w1018.state = STRAIGHT : 1;
      1 : 0;
      esac;

s3013_right := case
      r1.state = ACTIVE & w1018.state = RIGHT : 1;
      1 : 0;
      esac;

s3015_stop := case
      r4_lock_cond : 1;
      r5_lock_cond : 1;
      1 : 0;
      esac;

s3015_straight := case
      r4.state = ACTIVE & w1020.state = STRAIGHT : 1;
      1 : 0;
      esac;

s3015_right := case
      r5.state = ACTIVE & w1020.state = RIGHT : 1;
```

```
      1 : 0;
      esac;
```

7.6.4.4 Sensorverhalten

Nun folgt die Beschreibung des Sensorverhaltens.

7.6.4.4.1 Allgemeines Sensorverhalten In dem Modul `sensor` wird das Verhalten beschrieben, das für alle Sensoren gleich ist.
Als Parameter werden zwei Bedingungen übergeben, die angeben, wann der Sensor erreicht werden kann bzw. wann der Sensorzähler wieder zurückgesetzt werden soll.
Die Variable `ctr` repräsentiert den Sensorzähler. Anhand der Zähler kann man erkennen, wo sich momentan wie viele Bahnen befinden. Zudem dienen die Zähler dazu, zu bestimmen, wann eine Bahn einen bestimmten Sensor erreichen kann sowie zum Ausdrücken der Sicherheitsbedingungen.
Der Zähler wird inkrementiert, wenn die Bedingung `cond_1` (gibt an, ob der Sensor erreicht werden kann) erfüllt ist und wieder zurückgesetzt, wenn die Bedingung `cond_2` (gibt an, ob der Zähler zurückgesetzt werden soll) erfüllt ist.
Die Variable `state` repräsentiert den aktuellen Zustand des Sensors. Ein Sensor kann entweder im Zustand `HIGH` oder im Zustand `LOW` sein. `HIGH` drückt aus, dass der Sensor gerade erreicht wurde und `LOW`, dass der Sensor wieder frei ist und von einer folgenden Bahn befahren werden kann.

```
MODULE sensor(cond_1, cond_2)

VAR
state : {LOW, HIGH};
ctr : 0..3;

ASSIGN
init(state) := LOW;
init(ctr) := 0;

next(state) := case
      state = LOW & cond_1 : HIGH;
      state = HIGH : LOW;
      1 : state;
      esac;

next(ctr) := case
      state = LOW & cond_1 : ctr + 1;
      cond_2 : 0;
      1 : ctr;
      esac;
```

7.6.4.4.2 Initialisierung der Variablen Für die in einem Gleisnetz enthaltenen Sensoren werden Variablen erstellt. Ausgenommen sind dabei die Route Request Sensoren, die später gesondert behandelt werden. Es wird jeweils das Modul `sensor` aufgerufen und als Parameter werden die Bedingungen übergeben, die angeben, wann der Sensor erreichbar ist bzw. wann der Zähler zurückgesetzt werden soll.

```
g4125 : sensor(g4125_cond_1, g4125_cond_2);
g4126 : sensor(g4126_cond_1, g4126_cond_2);
g4127 : sensor(g4127_cond_1, g4127_cond_2);
g4128 : sensor(g4128_cond_1, g4128_cond_2);
g4129 : sensor(g4129_cond_1, g4129_cond_2);
g4130 : sensor(g4130_cond_1, g4130_cond_2);
```

7.6.4.4.3 Erreichbarkeitsbedingungen Wann ein Sensor erreichbar ist, ist von den aus der Beschreibung des Gleisnetzes gewonnenen Informationen abhängig, wo sich der jeweilige Sensor befindet. Ein Sensor kann erreicht werden, wenn sein Vorgängersensor passiert wurde und die Signal- und Weichenstellungen dies zulassen.
Befindet sich vor einem Sensor ein Signal, dann ist der Sensor nur erreichbar, wenn das Signal nicht auf `STOP` steht.
So ist in dem Beispiel der Sensor `g4125` nur erreichbar, wenn das Signal `s3013` nicht auf `STOP` steht. Damit der Sensor `g4125` erreichbar ist, muss zudem eine Bahn von Route Request Sensor 0 auf ihn zufahren.
Zudem ist es oft nur möglich, einen Sensor von einem anderen Sensor aus zu erreichen, wenn bestimmte Weichen bestimmte Stellungen haben.
So kann der Sensor `g4126` in dem Beispielgleisnetz erreicht werden, wenn die Weiche `w1018` nach rechts gestellt ist und eine Bahn von Sensor `g4125` auf ihn zufährt. Das eine Bahn von `g4125` auf `g4126` zufährt, kann man daran erkennen, dass der Zählerwert von `g4126` kleiner ist als der von `g4125`. Der Sensor `g4126` kann ebenfalls erreicht werden, wenn die Weiche `w1022` nach links gestellt ist und eine Bahn von Sensor `g4129` auf ihn zufährt.

```
g4125_cond_1 := case
      s3013.state != STOP & rr_0_next_tram != no : 1;
      1 : 0;
      esac;

g4126_cond_1 := case
      w1018.state = RIGHT & g4126.ctr < g4125.ctr : 1;
      w1022.state = LEFT & g4126.ctr < g4129.ctr : 1;
      1 : 0;
      esac;

g4127_cond_1 := case
      s3015.state != STOP & rr_2_next_tram != no : 1;
```

```
      1 : 0;
      esac;

g4128_cond_1 := case
      w1018.state = STRAIGHT & g4128.ctr < g4125.ctr : 1;
      w1020.state = RIGHT & g4128.ctr < g4127.ctr : 1;
      1 : 0;
      esac;

g4129_cond_1 := case
      s3014.state != STOP & rr_1_next_tram != no : 1;
      1 : 0;
      esac;

g4130_cond_1 := case
      w1020.state = STRAIGHT & g4130.ctr < g4127.ctr : 1;
      w1022.state = STRAIGHT & g4130.ctr < g4129.ctr : 1;
      1 : 0;
      esac;
```

7.6.4.4.4 Bedingungen zum Zurücksetzen der Zähler Der Zähler eines Sensors soll wieder zurückgesetzt werden, wenn eine Route, zu der der Sensor gehört, wieder in den Zustand `INACTIVE` übergeht.
Dies ist der Fall, wenn die Bedingung erfüllt ist, dass die Route in den Zustand `INACTIVE` wechseln soll und gleichzeitig nicht die Bedingung erfüllt ist, dass die Route in den Zustand `ACTIVE` wechseln soll. Wären beide Bedingungen erfüllt, dann würde die Route den Zustand `ACTIVE` annehmen. Dies kann passieren, wenn eine weitere Bahn die betreffende Route befahren möchte.
Der Zähler des Sensors `g4125` soll in dem Beispiel zurückgesetzt werden, wenn die Route `r0` oder die Route `r1` in den Zustand `INACTIVE` übergeht.

```
g4125_cond_2 := case
      r0_inactive_cond & !r0_active_cond : 1;
      r1_inactive_cond & !r1_active_cond : 1;
      1 : 0;
      esac;

g4126_cond_2 := case
      r1_inactive_cond & !r1_active_cond : 1;
      r3_inactive_cond & !r3_active_cond : 1;
      1 : 0;
      esac;

g4127_cond_2 := case
      r4_inactive_cond & !r4_active_cond : 1;
      r5_inactive_cond & !r5_active_cond : 1;
      1 : 0;
```

```
      esac;

g4128_cond_2 := case
      r0_inactive_cond & !r0_active_cond : 1;
      r5_inactive_cond & !r5_active_cond : 1;
      1 : 0;
      esac;

g4129_cond_2 := case
      r2_inactive_cond & !r2_active_cond : 1;
      r3_inactive_cond & !r3_active_cond : 1;
      1 : 0;
      esac;

g4130_cond_2 := case
      r2_inactive_cond & !r2_active_cond : 1;
      r4_inactive_cond & !r4_active_cond : 1;
      1 : 0;
      esac;
```

7.6.4.5 Routenverhalten

Jetzt wird beschrieben, wann eine Route welchen Zustand annehmen soll.

7.6.4.5.1 Allgemeines Routenverhalten Der Zustand einer Route wird durch die Variable `state` dargestellt, die die Werte `ACTIVE`, `LOCKED` und `INACTIVE` annehmen kann. Der Initialzustand jeder Route ist `INACTIVE`.
Es werden drei Bedingungen als Parameter übergeben. Die Bedingung `active_cond` gibt an, wann die Route den Zustand `ACTIVE` annehmen soll, die Bedingung `lock_cond`, wann die Route den Zustand `LOCKED` annehmen soll und die Bedingung `inactive_cond`, wann die Route den Zustand `INACTIVE` annehmen soll.
Dabei ist wichtig, dass die Anforderung, in den Zustand `ACTIVE` zu wechseln, eine höhere Priorität hat als die Anforderung, in den Zustand `INACTIVE` zu wechseln. Wenn also eine weitere Bahn die betreffende Route befahren möchte, wird ihr dies gestattet und die Route nimmt wieder den Zustand `ACTIVE` an.

```
MODULE route(active_cond, lock_cond, inactive_cond)

VAR
state : {INACTIVE, ACTIVE, LOCKED};

ASSIGN
init(state) := INACTIVE;

next(state) := case
      lock_cond : LOCKED;
```

```
      active_cond : ACTIVE;
      inactive_cond : INACTIVE;
      1 : state;
      esac;
```

7.6.4.5.2 Initialisierung der Variablen Für alle definierten Routen werden Variablen erstellt. Dazu wird jeweils das Modul `route` aufgerufen und als Parameter werden die drei erwähnten Bedingungen übergeben, die ausdrücken, wann eine Route den jeweiligen Zustand annehmen soll.

```
r0 : route(r0_active_cond, r0_lock_cond, r0_inactive_cond);
r1 : route(r1_active_cond, r1_lock_cond, r1_inactive_cond);
r2 : route(r2_active_cond, r2_lock_cond, r2_inactive_cond);
r3 : route(r3_active_cond, r3_lock_cond, r3_inactive_cond);
r4 : route(r4_active_cond, r4_lock_cond, r4_inactive_cond);
r5 : route(r5_active_cond, r5_lock_cond, r5_inactive_cond);
```

7.6.4.5.3 Routenzustand ACTIVE Eine Route soll den Zustand `ACTIVE` annehmen, wenn eine Bahn die Route angefordert hat und alle Konfliktrouten im Zustand `INACTIVE` sind. Dabei muss die betreffende Anforderung allerdings in der zu dem jeweiligen Route Request Sensor gehörigen Warteschlange an der ersten Position sein. Wenn zwei Konfliktrouten zum gleichen Zeitpunkt den Zustand `ACTIVE` annehmen könnten, wird die Route mit der höheren Priorität vorgezogen.
So befindet sich die Route `r0` in dem Beispiel im Konflikt mit den Routen `r1`, `r3`, `r4` und `r5`. Wenn eine Bahn bei Route Request Sensor 0 die Route `r0` angefordert hat und diese Anforderung in der Warteschlange, die zu Route Request Sensor 0 gehört, an der ersten Position ist, dann soll `r0` den Zustand `ACTIVE` annehmen, sofern die Konfliktrouten `r1`, `r3`, `r4` und `r5` im Zustand `INACTIVE` sind. Die Variable `rr_0_next_tram` steht dabei für die Anforderung, die sich an der ersten Position der zu Route Request Sensor 0 gehörigen Warteschlange befindet.

```
r0_active_cond := case
      rr_0_next_tram = r0 &
      r1.state = INACTIVE &
      r3.state = INACTIVE &
      r4.state = INACTIVE &
      r5.state = INACTIVE : 1;
      1 : 0;
      esac;

r1_active_cond := case
      rr_0_next_tram = r1 &
      r0.state = INACTIVE & !r0_active_cond &
      r3.state = INACTIVE : 1;
      1 : 0;
```

```
      esac;

r2_active_cond := case
      rr_1_next_tram = r2 &
      r3.state = INACTIVE &
      r4.state = INACTIVE : 1;
      1 : 0;
      esac;

r3_active_cond := case
      rr_1_next_tram = r3 &
      r0.state = INACTIVE & !r0_active_cond &
      r1.state = INACTIVE & !r1_active_cond &
      r2.state = INACTIVE & !r2_active_cond &
      r4.state = INACTIVE : 1;
      1 : 0;
      esac;

r4_active_cond := case
      rr_2_next_tram = r4 &
      r0.state = INACTIVE & !r0_active_cond &
      r2.state = INACTIVE & !r2_active_cond &
      r3.state = INACTIVE & !r3_active_cond &
      r5.state = INACTIVE : 1;
      1 : 0;
      esac;

r5_active_cond := case
      rr_2_next_tram = r5 &
      r0.state = INACTIVE & !r0_active_cond &
      r4.state = INACTIVE & !r4_active_cond : 1;
      1 : 0;
      esac;
```

7.6.4.5.4 Routenzustand LOCKED Eine Route soll in den Zustand `LOCKED` wechseln, wenn sie sich im Zustand `ACTIVE` befindet, der Eingangssensor der Route ausgelöst wurde und keine Bahn direkt folgt, die ebenfalls die betreffende Route befahren möchte.
In dem Beispiel soll die Route `r0` den Zustand `LOCKED` annehmen, wenn sie sich im Zustand `ACTIVE` befindet, der Eingangssensor `g4125` erreicht wurde und keine Bahn direkt folgt, die bei Route Request Sensor 0 ebenfalls die Route `r0` angefordert hat.

```
r0_lock_cond := case
      r0.state = ACTIVE & g4125.state = HIGH & !rr_0_next_tram = r0 : 1;
      1 : 0;
      esac;
```

```
r1_lock_cond := case
      r1.state = ACTIVE & g4125.state = HIGH & !rr_0_next_tram = r1 : 1;
      1 : 0;
      esac;

r2_lock_cond := case
      r2.state = ACTIVE & g4129.state = HIGH & !rr_1_next_tram = r2 : 1;
      1 : 0;
      esac;

r3_lock_cond := case
      r3.state = ACTIVE & g4129.state = HIGH & !rr_1_next_tram = r3 : 1;
      1 : 0;
      esac;

r4_lock_cond := case
      r4.state = ACTIVE & g4127.state = HIGH & !rr_2_next_tram = r4 : 1;
      1 : 0;
      esac;

r5_lock_cond := case
      r5.state = ACTIVE & g4127.state = HIGH & !rr_2_next_tram = r5 : 1;
      1 : 0;
      esac;
```

7.6.4.5.5 Routenzustand INACTIVE Eine Route soll in den Zustand `INACTIVE` wechseln, wenn sie im Zustand `LOCKED` ist, alle Signale, die sich auf der Route befinden, auf `STOP` sind und der Zähler des Eingangssensors den selben Wert hat wie der Zähler des Ausgangssensors, sich also keine Bahnen mehr auf der Route befinden.
In dem Beispiel soll demzufolge die Route `r0` den Zustand `INACTIVE` annehmen, wenn sie sich im Zustand `LOCKED` befindet, das Signal `s3013` auf `STOP` steht und der Eingangssensor `g4125` den selben Zählerwert hat wie der Ausgangssensor `g4128`.

```
r0_inactive_cond := case
      r0.state = LOCKED & s3013.state = STOP & g4125.ctr = g4128.ctr : 1;
      1 : 0;
      esac;

r1_inactive_cond := case
      r1.state = LOCKED & s3013.state = STOP & g4125.ctr = g4126.ctr : 1;
      1 : 0;
      esac;

r2_inactive_cond := case
      r2.state = LOCKED & s3014.state = STOP & g4129.ctr = g4130.ctr : 1;
      1 : 0;
      esac;
```

```
r3_inactive_cond := case
      r3.state = LOCKED & s3014.state = STOP & g4129.ctr = g4126.ctr : 1;
      1 : 0;
      esac;

r4_inactive_cond := case
      r4.state = LOCKED & s3015.state = STOP & g4127.ctr = g4130.ctr : 1;
      1 : 0;
      esac;

r5_inactive_cond := case
      r5.state = LOCKED & s3015.state = STOP & g4127.ctr = g4128.ctr : 1;
      1 : 0;
      esac;
```

7.6.4.6 Routenanforderungen

Jedem Route Request Sensor werden eine Reihe von Routen zugeordnet, die an diesem Route Request Sensor angefordert werden können. Zudem wird jedem Route Request Sensor eine Warteschlange zugeordnet, die die eingegangenen Routenanforderungen speichert. Hier wird jeweils die Anforderung ermittelt, die sich an der ersten Position der betreffenden Warteschlange befindet.
In dem Beispielgleisnetz gibt es drei Route Request Sensoren. An dem Route Request Sensor 0 können z.B. die Routen `r0` und `r1` angefordert werden.

```
VAR
rr_0_next_tram : {no, r0, r1};
rr_1_next_tram : {no, r2, r3};
rr_2_next_tram : {no, r4, r5};

ASSIGN
init(rr_0_next_tram) := no;
init(rr_1_next_tram) := no;
init(rr_2_next_tram) := no;

next(rr_0_next_tram) := case
      g4125.ctr >= 3 : no;
      rr_0_next_tram = no : {no, r0, r1};
      g4125.state = LOW & g4125_cond_1 : {no, r0, r1};
      1 : rr_0_next_tram;
      esac;

next(rr_1_next_tram) := case
      g4129.ctr >= 3 : no;
      rr_1_next_tram = no : {no, r2, r3};
      g4129.state = LOW & g4129_cond_1 : {no, r2, r3};
```

```
      1 : rr_1_next_tram;
      esac;

next(rr_2_next_tram) := case
      g4127.ctr >= 3 : no;
      rr_2_next_tram = no : {no, r4, r5};
      g4127.state = LOW & g4127_cond_1 : {no, r4, r5};
      1 : rr_2_next_tram;
      esac;
```

7.6.4.7 Sicherheitsbedingungen

Die Sicherheitsbedingungen werden als Invariante spezifiziert. Das bedeutet, es wird geprüft, ob:

- (1) im Startzustand alle Sicherheitsbedingungen erfüllt sind
- (2) im Zustand p+1 alle Sicherheitsbedingungen unter der Annahme erfüllt sind, dass sie im Zustand p gelten

```
INVARSPEC safe
```

Die Invariante `safe` beinhaltet alle für das jeweilige Gleisnetz geltenden Sicherheitsanforderungen. Ist die Invariante wahr, dann bedeutet es demzufolge, dass alle Sicherheitsanforderungen erfüllt sind. Ist die Invariante falsch, wird ein Gegenbeispiel erzeugt. In dem Gegenbeispiel wird ein Zustand erzeugt, in dem eine Sicherheitsbedingung nicht erfüllt ist.

```
safe := sr1 & sr2 & sr3 & sr4 & sr5;
```

Die Invariante `safe` beinhaltet also fünf Bedingungen, die folgende Bedeutungen haben:

- `sr1` enthält alle Sicherheitsbedingungen, die eingehalten werden müssen, damit keine Bahnen aus unterschiedlichen Richtungen auf dieselbe Kreuzung zufahren, so dass es zu einem Flankenzusammenstoß kommen kann.
- `sr2` enthält alle Sicherheitsbedingungen, die eingehalten werden müssen, damit keine Bahnen aus unterschiedlichen Richtungen auf dieselbe Weiche zufahren, so dass es zu einem Flankenzusammenstoß kommen kann.
- `sr3` enthält alle Sicherheitsbedingungen, die eingehalten werden müssen, damit keine Bahnen auf Weichen auffahren, wenn das Auffahren nicht möglich ist.
- `sr4` enthält alle Sicherheitsbedingungen, die eingehalten werden müssen, damit keine Weichen umgeschaltet werden, während sie befahren werden.

- `sr5` beinhaltet alle Sicherheitsbedingungen, die eingehalten werden müssen, damit keine Bahnen in entgegengesetzter Fahrtrichtung aufeinander zufahren, so dass es zu einem Frontalzusammenstoß kommen kann.

Jede dieser fünf Bedingungen beinhaltet die Sicherheitsbedingungen der jeweiligen Kategorie, die in dem konkreten Gleisnetz gelten müssen.

7.6.4.7.1 Erste Sicherheitsbedingung Ist eine Kreuzung von zwei unterschiedlichen Sensoren erreichbar, dann dürfen die Zählerwerte dieser beiden Sensoren nicht beide größer als 0 sein. Damit eine Kreuzung von einem Sensor erreichbar ist, sind häufig bestimmte Weichenstellungen notwendig.
In dem Beispielgleisnetz ist die Kreuzung `c2016` von Sensor `g4125` erreichbar, wenn die Weiche `w1018` geradeaus gestellt ist und von Sensor `g4129`, wenn die Weiche `w1022` nach links gestellt ist. Demzufolge darf kein Zustand auftreten, in dem die Zähler der Sensoren `g4125` und `g4129` beide größer als 0 sind, die Weiche `w1018` geradeaus gestellt ist und die Weiche `w1022` nach links gestellt ist. Dies wird durch die Sicherheitsbedingung `sr1_1` ausgedrückt. Wäre ein solcher Zustand erreichbar, dann wäre das System nicht sicher.

```
sr1 := sr1_1 & sr1_2 & sr1_3;

sr1_1 := !(g4125.ctr > 0 & w1018.state = STRAIGHT &
           w1022.state = LEFT & g4129.ctr > 0);

sr1_2 := !(g4125.ctr > 0 & w1018.state = STRAIGHT &
           w1020.state = STRAIGHT & g4127.ctr > 0);

sr1_3 := !(g4129.ctr > 0 & w1022.state = LEFT &
           w1020.state = STRAIGHT & g4127.ctr > 0);
```

7.6.4.7.2 Zweite Sicherheitsbedingung Ist eine Weiche von zwei unterschiedlichen Sensoren erreichbar, dann dürfen die Zählerwerte dieser beiden Sensoren ebenfalls nicht beide größer als 0 sein.
So ist in dem Beispielgleisnetz die Weiche `w1019` von Sensor `g4129` ereichbar, wenn die Weiche `w1022` nach links gestellt ist und von Sensor `g4125`, wenn die Weiche `w1018` nach rechts gestellt ist. Daher darf kein Zustand auftreten, in dem die Zähler der Sensoren `g4125` und `g4129` größer als 0 sind, die Weiche `w1022` nach links gestellt ist und die Weiche `w1018` nach rechts gestellt ist. Dies wird durch die Sicherheitsbedingung `sr2_1` ausgedrückt.

```
sr2 := sr2_1 & sr2_2 & sr2_3;

sr2_1 := !(g4129.ctr > 0 & w1022.state = LEFT &
           w1018.state = RIGHT & g4125.ctr > 0);
```

```
sr2_2 := !(g4125.ctr > 0 & w1018.state = STRAIGHT &
           w1020.state = RIGHT & g4127.ctr > 0);

sr2_3 := !(g4127.ctr > 0 & w1020.state = STRAIGHT &
           w1022.state = STRAIGHT & g4129.ctr > 0);
```

7.6.4.7.3 Dritte Sicherheitsbedingung Wenn eine Weiche von einem Sensor erreichbar ist und sich eine Bahn an dem Sensor befindet, dann darf es nicht sein, dass die Weiche nicht auffahrbar ist.
Eine aktive Weiche ist nicht auffahrbar, wenn sie noch nicht gestellt wurde. Außerdem kann es sein, dass ein Befahren von einer nicht gestellten stumpfen Seite nicht möglich ist.
Für das Beispiel wird u.a. eine Bedingung erstellt, dass, wenn eine Bahn bereits den Sensor `g4129` passiert hat oder der Sensor erreichbar ist, sich die Weiche `w1022` nicht mehr im Zustand `UNDEFINED` befinden darf.

```
sr3 := sr3_1 & sr3_2 & sr3_3;

sr3_1 := ((g4129.ctr > 0 | g4129_cond_1) -> w1022.state != UNDEFINED);

sr3_2 := ((g4125.ctr > 0 | g4125_cond_1) -> w1018.state != UNDEFINED);

sr3_3 := ((g4127.ctr > 0 | g4127_cond_1) -> w1020.state != UNDEFINED);
```

7.6.4.7.4 Vierte Sicherheitsbedingung Wenn eine Weiche von einem Sensor erreichbar ist und sich eine Bahn an dem Sensor befindet, dann darf es nicht sein, dass eine Anforderung zum Umschalten der Weiche besteht.
Wenn sich in dem Beispiel eine Bahn an Sensor `g4129` befindet, darf keine Anforderung eingehen, die Weiche `w1022` umzuschalten.

```
sr4 := sr4_1 & sr4_2 & sr4_3;

sr4_1 := ((g4129.ctr > 0 | g4129_cond_1) & w1022.state = STRAIGHT -> !w1022_left) &
         ((g4129.ctr > 0 | g4129_cond_1) & w1022.state = LEFT -> !w1022_straight);

sr4_2 := ((g4125.ctr > 0 | g4125_cond_1) & w1018.state = STRAIGHT -> !w1018_right) &
         ((g4125.ctr > 0 | g4125_cond_1) & w1018.state = RIGHT -> !w1018_straight);

sr4_3 := ((g4127.ctr > 0 | g4127_cond_1) & w1020.state = STRAIGHT -> !w1020_right) &
         ((g4127.ctr > 0 | g4127_cond_1) & w1020.state = RIGHT -> !w1020_straight);
```

7.6.4.7.5 Fünfte Sicherheitsbedingung Wenn ein Abschnitt des Gleisnetzes von zwei Eingangssensoren (Sensoren, die von einem Route Request Sensor erreichbar sind) aus entgegengesetzten Richtungen erreichbar ist, dürfen die Zählerwerte dieser beiden Sensoren nicht beide größer als 0 sein.

Diese Sicherheitsbedingung muss nur beachtet werden, wenn es in einem Gleisnetz möglich ist, dass ein Abschnitt in beide Richtungen befahren wird. In dem Beispielgleisnetz ist dies nicht der Fall und daher kann diese Sicherheitsbedingung hier nicht verletzt sein.

```
sr5 := 1;
```

7.6.4.8 Safety Monitor

Für den Safety Monitor wird eine Menge von Zuständen definiert, die als sicher betrachtet werden. Die Sicherheitsbedingungen wurden bekanntlich aus der Beschreibung des Gleisnetzes abgeleitet. Im Gegensatz dazu werden diese Zustände aus den in den Verschlusstabellen gemachten Angaben abgeleitet. Wenn die dort gemachten Angaben nicht sicher sind, sind demzufolge diese Zustände auch nicht sicher.
In dem Beispiel werden 14 Zustände als sicher angesehen.

```
st0 := s3013.state = STOP & g4125.ctr = 0 &
       g4129.ctr >= g4126.ctr & w1022.state = LEFT &
       g4127.ctr >= g4128.ctr & w1020.state = RIGHT;

st1 := g4125.ctr >= g4126.ctr & w1018.state = RIGHT &
       g4129.ctr >= g4130.ctr & w1022.state = STRAIGHT &
       g4127.ctr >= g4128.ctr & w1020.state = RIGHT;

st2 := s3013.state = STOP & g4125.ctr = 0 &
       g4129.ctr >= g4130.ctr & w1022.state = STRAIGHT &
       g4127.ctr >= g4128.ctr & w1020.state = RIGHT;

st3 := g4125.ctr >= g4126.ctr & w1018.state = RIGHT &
       s3014.state = STOP & g4129.ctr = 0 &
       g4127.ctr >= g4128.ctr & w1020.state = RIGHT;

st4 := s3013.state = STOP & g4125.ctr = 0 &
       s3014.state = STOP & g4129.ctr = 0 &
       g4127.ctr >= g4128.ctr & w1020.state = RIGHT;

st5 := g4125.ctr >= g4126.ctr & w1018.state = RIGHT &
       s3014.state = STOP & g4129.ctr = 0 &
       g4127.ctr >= g4130.ctr & w1020.state = STRAIGHT;

st6 := s3013.state = STOP & g4125.ctr = 0 &
       s3014.state = STOP & g4129.ctr = 0 &
       g4127.ctr >= g4130.ctr & w1020.state = STRAIGHT;

st7 := s3013.state = STOP & g4125.ctr = 0 &
       g4129.ctr >= g4126.ctr & w1022.state = LEFT &
       s3015.state = STOP & g4127.ctr = 0;
```

```
st8 := g4125.ctr >= g4126.ctr & w1018.state = RIGHT &
       g4129.ctr >= g4130.ctr & w1022.state = STRAIGHT &
       s3015.state = STOP & g4127.ctr = 0;

st9 := g4125.ctr >= g4128.ctr & w1018.state = STRAIGHT &
       g4129.ctr >= g4130.ctr & w1022.state = STRAIGHT &
       s3015.state = STOP & g4127.ctr = 0;

st10 := s3013.state = STOP & g4125.ctr = 0 &
        g4129.ctr >= g4130.ctr & w1022.state = STRAIGHT &
        s3015.state = STOP & g4127.ctr = 0;

st11 := g4125.ctr >= g4126.ctr & w1018.state = RIGHT &
        s3014.state = STOP & g4129.ctr = 0 &
        s3015.state = STOP & g4127.ctr = 0;

st12 := g4125.ctr >= g4128.ctr & w1018.state = STRAIGHT &
        s3014.state = STOP & g4129.ctr = 0 &
        s3015.state = STOP & g4127.ctr = 0;

st13 := s3013.state = STOP & g4125.ctr = 0 &
        s3014.state = STOP & g4129.ctr = 0 &
        s3015.state = STOP & g4127.ctr = 0;
```

Es wird eine Invariante `safety_monitor` definiert, die erfüllt ist, wenn einer der 14 sicheren Zustände gilt.

```
safety_monitor := st0 | st1 | st2 | st3 | st4 | st5 | st6 | st7 |
                  st8 | st9 | st10 | st11 | st12 | st13;

INVARSPEC safety_monitor
```

7.6.4.9 Ergebnisse

Die hier beschriebene Verifikation wurde für unsere Teststrecke durchgeführt. Die Teststrecke ist im Abschnitt 7.7.4 auf Seite 262 zu finden.
Dass alle Sicherheitsbedingungen erfüllt sind, lässt sich für die Strecken 2, 3 und 4 jeweils innerhalb weniger Sekunden nachweisen. Für die komplexere Strecke 1 braucht der Model Checker dazu rund fünf Minuten. Dabei wurde ein Pentium 4 Prozessor mit 2.6 GHz verwendet.
Wenn man das Programm so verändert, dass eine Sicherheitsbedingung nicht erfüllt ist, wird dies für jede der vier Strecken vom Model Checker innerhalb einiger Sekunden erkannt und ein entsprechendes Gegenbeispiel erzeugt.

7.6.5 Automatische Generierung der Eingabedatei

Damit das System mit Model Checking automatisch verifiziert werden kann, wird die Eingabedatei des Model Checkers für das jeweilige zu steuernde Gleisnetz automatisch generiert. Dazu werden bestimmte Informationen verwendet, die benötigt werden, um das System zu modellieren. Zum Erstellen des physikalischen Modells werden Informationen aus der TND-Gleisnetzbeschreibung verwendet und zum Erstellen des Controller Modells Informationen aus den Projektierungsdaten. Die Sicherheitsbedingungen lassen sich ebenfalls anhand der Beschreibung des betrachteten Gleisnetzes automatisch erzeugen.

7.6.5.1 Physikalisches Modell

Um das physikalische Modell zu erstellen, werden Informationen über die Beschaffenheit des Gleisnetzes aus der TND benötigt. Diese Informationen sind in folgenden Blöcken enthalten.

- definitions-Block
- point-properties-Block
- relations-Block
- signal-positions-Block

7.6.5.1.1 definitions-Block Der definitions-Block beinhaltet alle im Gleisnetz enthaltenen Sensoren, Weichen, Signale und Kreuzungen. Zunächst benötigt man für ein Gleisnetz folgende Informationen:

- Informationen über die Menge der Signale S_{set}

  Diese Menge ist definiert durch die Funktion $signals_{set}()$:

  $$signals_{set} : tnd \longrightarrow S_{set}$$

  Diese Funktion liefert die Menge aller in der tnd enthaltenen Signale.

  Analog zur Funktion $signals_{set}()$ werden die Funktionen $points_{set}()$, $sensors_{set}()$ und $crossings_{set}()$ definiert.

- Informationen über die Menge der Weichen P_{set}

  $$points_{set} : tnd \longrightarrow P_{set}$$

- Informationen über die Menge der Sensoren G_{set}

$$sensors_{set} : tnd \longrightarrow G_{set}$$

- Informationen über die Menge der Kreuzungen K_{set}

$$crossings_{set} : tnd \longrightarrow K_{set}$$

Zudem muss man wissen, von welchem Typ die jeweilige Weiche bzw. das jeweilige Signal ist und ob der jeweilige Sensor ein Route Request Sensor ist. Dazu werden die folgenden Funktionen umgesetzt.

- Typ des Signals

  Die Funktion $signal_{typ}()$ ist wie folgt definiert:

$$signal_{typ} : (s \in S_{set}) \longrightarrow \{S, L, R, SL, SR, LR, SLR\}$$

  Sie gibt für ein gegebenes Signal s aus der TND den Typ des Signals zurück. Mit S, L, R sind hier straight, left bzw. right gemeint und SR, LR, SLR repräsentieren die entsprechenden Kombinationen.

- Typ der Weiche

  Analog zu der Funktion $signal_{typ}()$ ist die Funktion $point_{typ}()$ folgendermaßen definiert:

$$point_{typ} : (p \in P_{set}) \longrightarrow \{SL, SR, LR, SLR\}$$

- Typ des Sensors

  Da ein Sensor entweder ein Route Request Sensor oder ein anderer Sensor ist, wird die Funktion $is\_rrsensor()$ als Boolean definiert:

$$is\_rrsensor : (g \in G_{set}) \longrightarrow Bool$$

  Diese Funktion liefert für einen gegebenen Sensor g $true$ zurück, falls g ein Route Request Sensor ist. Ansonsten wird $false$ zurückgegeben.

- Kreuzung

  Es gibt nur einen Typ von Kreuzungen. Daher wird keine Funktion benötigt, die den Typ der Kreuzung liefert.

Für das Beispiel sieht der definitions-Block folgendermaßen aus:

```
definitions {
   rr-sensors: g4023, g4025, g4027;
   tg-sensors: g4125, g4126, g4127, g4128, g4129, g4130;
   sl-points: w1019, w1021, w1022;
   sr-points: w1018, w1020, w1023;
   sl-signals: s3014;
   sr-signals: s3013, s3015;
   ..........
   crossings: k2016, k2017, k2018;
}
```

7.6.5.1.2 relations-Block Im relations-Block werden die Nachbarschaftsbeziehungen der einzelnen Elemente definiert. Daraus kann man ableiten, welches Element von welchem anderen Element aus erreichbar ist.
Die Funktion $is\_reachable()$ prüft, ob ein bestimmtes Element von einem bestimmten Sensor erreichbar ist, wenn man von dem Sensor in eine bestimmte Richtung fährt. Diese Funktion ist wie folgt definiert:

$$is\_reachable : (x \in (P_{set} \cup G_{set} \cup K_{set})) \times (g \in G_{set}) \times (y \in \{A, B\}) \longrightarrow \{-1, (r \times [p'])\}$$

mit $r \in \{A, B, Y, L, R, S, C, D\}$ und $p' \in P_{setting}$

$P_{setting}$ ist die Menge aller möglichen Einstellungen der jeweiligen Weichen. $[p']$ bedeutet, dass die Komponente p' für das Paar $(r \times [p'])$ optional ist.
Wie für die TND definiert, sind A und B die Seiten eines Sensors, Y, L, R und S die Seiten einer Weiche und A, B, C und D die Seiten einer Kreuzung.
Ist das betreffende Element von dem betreffenden Sensor in der betreffenden Richtung erreichbar, wird die Seite zurückgegeben, an der das Element erreicht werden kann. Zudem kann es sein, dass das Element nur erreichbar ist, wenn bestimmte Weichenstellungen vorliegen. Diese Weichenstellungen werden ebenfalls ermittelt. Wenn das Element von dem Sensor nicht erreichbar ist, dann wird -1 zurückgegeben.
Es soll nun beispielhaft angegeben werden, wie geprüft wird, ob der Sensor g4126 vom Sensor g4125 aus erreichbar ist.
Wenn man von Sensor g4126 in Richtung A geht, gelangt man zu der Seite B des Sensors g4023. Daher ist in dieser Richtung der Sensor g4126 kein direkter Folgesensor von g4125. Wenn man von Sensor g4126 in Richtung B geht, gelangt man zu der Seite Y der Weiche w1018. Von dort kann man geradeaus oder nach rechts weitergehen. Wenn man geradeaus weitergeht, ist der Sensor g4126 nicht erreichbar. Geht man nach rechts weiter, wird zunächst die Seite L der Weiche w1019 erreicht. Von Seite Y der Weiche w1019 erreicht man Seite A des Sensors g4126. Der Sensor ist also von Sensor g4125 zu erreichen und zwar auf Seite A. Eine Bedingung dafür ist allerdings, dass die Weiche w1018 nach rechts gestellt ist.

```
is_reachable(g4126, g4125, A)

g4125 A -> g4023 B
return -1

-----------------------------------------------------------------
is_reachable(g4126, g4125, B)

                        g4125 B -> w1018 Y
                       /                  \
w1018 R -> w1019 L                            w1018 S -> k2016 A
w1019 Y -> g4126 A                            k2016 B -> k2017 A
return (A, w1018.state = RIGHT)               k2017 B -> w1021 S
                                              w1021 Y -> g4128 A
                                              return -1
```

Der relations-Block sieht für das Beispiel folgendermaßen aus:

```
relations {
   g4023 A: entrance;
   g4025 A: entrance;
   g4027 A: entrance;
   g4023 B: x1, g4125 A;
   g4025 B: x2, g4129 A;
   g4027 B: x3, g4127 A;
   g4126 B: exit;
   g4128 B: exit;
   g4130 B: exit;
   g4125 B: w1018 Y;
   w1019 Y: g4126 A;
   g4127 B: w1020 Y;
   w1021 Y: g4128 A;
   g4129 B: w1022 Y;
   w1023 Y: g4130 A;
   w1018 R: w1019 L;
   w1020 R: w1021 L;
   w1022 S: w1023 S;
   w1018 S: k2016 A;
   k2016 D: w1019 S;
   k2017 B: w1021 S;
   w1020 S: k2017 C;
   w1022 L: k2018 A;
   k2018 D: w1023 R;
   k2016 B: k2017 A;
   k2018 B: k2016 C;
   k2017 D: k2018 C;
}
```

7.6.5.1.3 point-properties-Block Im point-properties-Block sind zusätzliche Eigenschaften der Weichen definiert.
Der Block sieht für das Beispiel so aus:

```
point-properties {
   w1018: switchtime 800;
   w1019: passive switchtime 800;
   w1020: switchtime 800;
   w1021: passive switchtime 800;
   w1022: switchtime 800;
   w1023: passive switchtime 800;
}
```

Die Funktion *active*() gibt an, ob die jeweilige Weiche eine Aktiv- oder eine Passivweiche ist und *breakable*() bzw. $fallback()$ geben an, ob die Weiche die Eigenschaft BREAKABLE bzw. FALLBACK besitzt. Hat die Weiche die Eigenschaft FALLBACK, dann gibt $fallback\_direction()$ die vordefinierte Richtung zurück.
Diese Funktionen sind wie folgt definiert:

$$active : (p \in P_{set}) \longrightarrow Bool$$

$$breakable : (p \in P_{set}) \longrightarrow Bool$$

$$fallback : (p \in P_{set}) \longrightarrow Bool$$

$$fallback\_direction : (p \in P_{set}) \longrightarrow Integer$$

7.6.5.1.4 signal-positions-Block Im signal-positions-Block wird definiert, wo sich die Signale befinden.
Dies sieht für das Beispielgleisnetz folgendermaßen aus:

```
signal-positions  {
   s3013: g4023 B - x1 - g4125 A;
   s3014: g4025 B - x2 - g4129 A;
   s3015: g4027 B - x3 - g4127 A;
}
```

Die folgende Funktion gibt für ein Signal $s \in S_{set}$ und einen Sensor $g \in G_{set}$ an, ob sich das Signal an der Seite *side* vor dem Sensor befindet.

$$signal\_before : ((s \in S_{set}) \times (g \in G_{set}) \times (side \in \{A, B\})) \longrightarrow Bool$$

So befindet sich in dem Beispiel das Signal s3013 an der Seite A des Sensors g4125:

$$signal\_before(s3013, g4125, A) : TRUE$$

7.6.5.2 Controller Modell

Um das Controller Modell zu erstellen, werden Informationen über die Routen aus den Projektierungsdaten benötigt.

7.6.5.2.1 Route Dispatcher Aus den Projektierungsdaten für den Route Dispatcher werden die Informationen benötigt, die sich durch folgende Funktionen ausdrücken lassen.

- Die Funktion $routes_{set}()$

  $$routes_{set} : Projektierungsdaten \longrightarrow R_{set}$$

  liefert die Menge aller in den Projektierungsdaten enthaltenen Routen.

- Die Funktion $conflict()$

  $$conflict : ((route1 \in R_{set}) \times (route2 \in R_{set})) \longrightarrow Bool$$

  gibt für zwei unterschiedliche Routen $route1 \in R_{set}$ und $route2 \in R_{set}$ an, ob ein Konflikt zwischen ihnen definiert ist.

- Die Funktion $rrsensor\_id()$

  $$rrsensor\_id : (route \in R_{set}) \longrightarrow Integer$$

  liefert die ID des Route Request Sensors einer Route $route \in R_{set}$.

7.6.5.2.2 Route Controller Aus den Projektierungsdaten für den Route Controller werden die Informationen benötigt, die sich durch folgende Funktionen ausdrücken lassen.
Welche Signal- bzw. Weichenstellungen jeweils benötigt werden, kann man daran erkennen, welches Bit gesetzt ist.

- Die Funktion $point\_request()$

  $$point\_request : ((p \in P_{set}) \times (route \in R_{set}) \times direction) \longrightarrow Bool$$

  mit $direction \in \{L, R, S\}$

  gibt an, ob eine Weiche $p \in P_{set}$ für eine Route $route \in R_{set}$ in die angegebene Richtung $direction$ gestellt werden soll.

- Die Funktion $signal\_free()$

$$signal\_free : ((s \in S_{set}) \times (route \in R_{set}) \times direction) \longrightarrow Bool$$

  mit $direction \in \{L, R, S\}$

  gibt an, ob ein Signal $s \in S$ für eine Route $route \in R_{set}$ in die angegebene Richtung $direction$ gestellt werden soll.

- Die Funktion signal_lock()

$$signal\_lock : ((s \in S_{set}) \times (route \in R_{set})) \longrightarrow Bool$$

  gibt an, ob ein Signal $s \in S_{set}$ auf STOP gesetzt werden soll, wenn eine Route $route \in R_{set}$ wieder gesperrt wird.

- Die Funktionen $entrance\_sensor()$ und $exit\_sensor()$

$$entrance\_sensor : (route \in R_{set}) \longrightarrow (g \in G_{set})$$

$$exit\_sensor : (route \in R_{set}) \longrightarrow (g \in G_{set})$$

  liefern den Eingangs- bzw. den Ausgangssensor einer Route $route \in R$.

Alle oben beschriebenen Funktionen wurden in c umgesetzt. Abhängig davon, welche Informationen diese Funktionen liefern, wird die Model Checker Eingabedatei für das betreffende Gleisnetz generiert.

Ruben Rothaupt, Taffou Happi

7.6.6 Überprüfung der Verschlusstabellen

In diesem Abschnitt soll beschrieben werden, wie die in den Verschlusstabellen gemachten Angaben überprüft werden können. Die Verschlusstabellen bestehen aus Route Definition Table, Point Position Table, Signal Setting Table und Route Conflict Table. Die dort enthaltenen Informationen sind in der TND in folgenden Blöcken umgesetzt:

- Route Definition Table
  - routedefinitions-Block
- Point Position Table
  - conditions-Block
- Signal Setting Table

  - clearances-Block

- Route Conflict Table
  - conflicts-Block
  - point-conflicts-Block

Diese fünf Blöcke müssen also überprüft werden.
Im routedefinitions-Block wird für jede Route definiert, in welcher Reihenfolge welche Sensoren passiert werden sollen. Der routedefinitions-Block kann nun überprüft werden, indem alle möglichen Routen einzeln befahren werden. Ein Fehler wird erkannt, wenn die Sensoren nicht in der definierten Reihenfolge passiert werden. Dies würde bedeuten, dass die jeweilige Route nicht so befahrbar ist, wie sie definiert wurde.
Beim Überprüfen des routedefinitions-Blocks wird auch gleichzeitig überprüft, ob die im conditions-Block enthaltenen Angaben richtig sind. Der conditions-Block enthält die für eine Route benötigten Weichenstellungen. Wäre dort eine Weichenstellung für eine Route falsch definiert, könnte die betreffende Route ebenfalls nicht in der beabsichtigten Weise durchfahren werden.
Wenn alle Routendefinitionen als richtig erkannt werden, dann wurde gezeigt, dass im routedefinitions-Block und im conditions-Block keine Fehler enthalten sind.
Es soll geprüft werden, dass conditions-Block und clearances-Block keine Widersprüche enthalten. Befindet sich ein Signal direkt an der spitzen Seite einer aktiven Weiche, dann darf es nicht möglich sein, dass sich für eine Route die geforderten Richtungen für das Signal und die Weiche unterscheiden. So soll es z.B. nicht möglich sein, dass für eine Route die Weiche nach links und das Signal nach rechts gestellt werden soll.
Wenn keine Widersprüche zwischen Signal- und Weichenanforderungen bestehen und alle Routendefinitionen als richtig erkannt wurden, dann wurde auch gezeigt, dass im clearances-Block keine Fehler enthalten sind.
Im conflicts-Block wird für zwei Routen ein Konflikt definiert, wenn ein gemeinsames Befahren dieser beiden Routen zu einer Kollision führen könnte. Der conflicts-Block kann überprüft werden, indem alle möglichen Zweierpaare von Routen befahren werden.
Wenn für die jeweilige Routenkombination kein Konflikt definiert ist, dann darf auch keine Kollision möglich sein, wenn die beiden betreffenden Routen gleichzeitig befahren werden. Ansonsten wäre vergessen worden, für diese Routenkombination einen Konflikt zu definieren.
Wenn ein Konflikt definiert wurde, dann muss für die betreffende Routenkombination eine Kollision möglich sein. Wäre keine Kollision möglich, dann wäre die Definition eines Konfliktes hier falsch, da die betreffenden Routen nicht im Konflikt stehen würden.
Sind diese beiden Bedingungen erfüllt, dann sind die im conflicts-Block enthaltenen Angaben korrekt.

Entsprechend zum conflicts-Block kann auch der point-conflicts-Block geprüft werden. Im point-conflicts-Block werden mögliche Weichenkonflikte definiert. Ein Weichenkonflikt besteht, wenn für eine Weiche für unterschiedliche Routen unterschiedliche Weichenstellungen angefordert werden. Ein Weichenkonflikt führt also ebenfalls dazu, dass die betreffenden Routen nicht gleichzeitig befahren werden dürfen.
Wenn für eine Routenkombination ein Weichenkonflikt definiert wurde, muss dieser auch tatsächlich existieren. Wurde kein Weichenkonflikt definiert, dann darf auch keiner möglich sein.
Gelten diese beiden Bedingungen, dann sind die im point-conflicts-Block enthaltenen Angaben korrekt.
Im Folgenden soll nun das Modell zum Prüfen der Verschlusstabellen erläutert werden. Dazu soll wiederum die Teilstrecke 4 der Teststrecke als Beispiel dienen, die in Abbildung 7.48 angegeben ist.

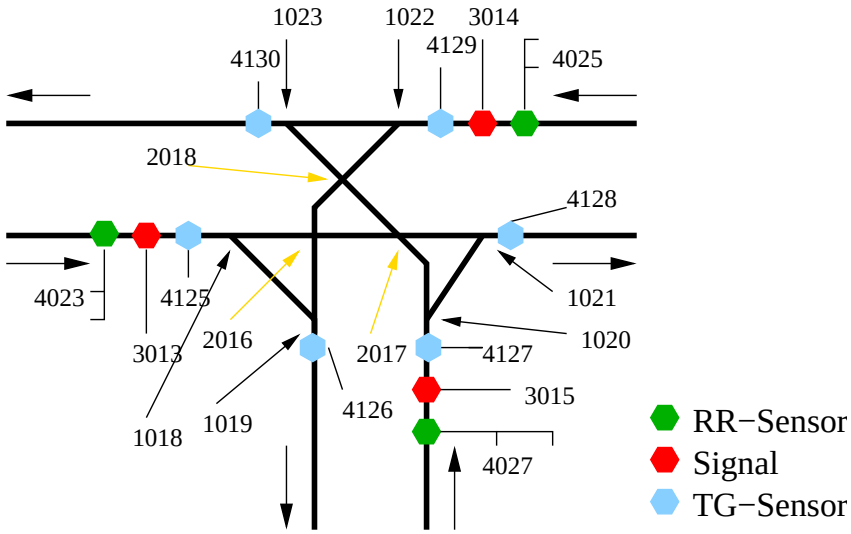


Abbildung 7.48: Strecke 4

Route Definition Table

Route	Route Sensor Sequence	Route Request Sensor
route23	(g4125, g4128)	g4023
route24	(g4125, g4126)	g4023
route25	(g4129, g4130)	g4025
route26	(g4129, g4126)	g4025
route27	(g4127, g4130)	g4027
route28	(g4127, g4128)	g4027

Point Position Table

Route	w1018	w1020	w1022
route23	straight	-	-
route24	right	-	-
route25	-	-	straight
route26	-	-	left
route27	-	straight	-
route28	-	right	-

Signal Setting Table

Route	Signal	Setting
route23	s3013	straight
route24	s3013	right
route25	s3014	straight
route26	s3014	left
route27	s3015	straight
route28	s3015	right

Route Conflict Table

Route	route23	route24	route25	route26	route27	route28
route23		**X**		X	X	X
route24	**X**			X		
route25				**X**	X	
route26	X	X	**X**		X	
route27	X		X	X		**X**
route28	X				**X**	

7.6.6.1 Routenanforderungen

Zur Überprüfung der Verschlusstabellen müssen nur maximal zwei Routen befahren werden. Dadurch kann man feststellen, ob zwischen diesen beiden Routen ein Konflikt besteht. Dass sich mehrere Bahnen auf einer Route befinden können oder dass sich Routen in bestimmten Zuständen befinden, ist für die Überprüfung der Verschlusstabellen nicht notwendig.
Zum Überprüfen des Route Conflicts Table müssen alle Routenkombinationen durchprobiert werden. Ist beispielsweise `route_A=route23` und `route_B=route26`, dann werden die beiden Routen `route23` und `route26` befahren. Nun kann man erkennen, ob diese Routen im Konflikt stehen.
Zum Überprüfen von Route Definition Table, Point Position Table und Signal Setting Table muss jeweils nur eine einzelne Route befahren werden statt eines Routenpaares. Wenn man nun `route_A=route23` und `route_B=no` setzt, dann wird nur die Route `route23` befahren. Jetzt kann man erkennen, ob die Route so durchfahren wird, wie es definiert wurde.
Es fällt auf, dass die Routen hier andere Namen haben, als bei der Überprüfung des

Systemmodells, obwohl es sich um das selbe Beispielgleisnetz handelt. Dies liegt daran, dass bei der Verschlusstabellenüberprüfung die Informationen und damit auch die Routennnamen aus der TND gewonnen werden, während die Routeninformationen bei der Überprüfung des Systemmodells aus den Projektierungsdaten kommen.

```
VAR
route_A : {route23, route24, route25, route26, route27, route28, no};
route_B : {route23, route24, route25, route26, route27, route28, no};

ASSIGN
init(route_A) := {route23, route24, route25, route26, route27, route28, no};
init(route_B) := {route23, route24, route25, route26, route27, route28, no};

next(route_A) := no;
next(route_B) := no;
```

7.6.6.2 Weichenverhalten

Zunächst wird die Modellierung des Weichenverhaltens beschrieben.

7.6.6.2.1 Allgemeines Weichenverhalten In dem Modul `point` wird das Verhalten beschrieben, das für alle Weichen gleich ist. Als Parameter werden drei Bedingungen übergeben, die angeben, ob eine Anforderung existiert, die entsprechende Weiche geradeaus, nach links oder nach rechts zu stellen.
Die Variable `state` repräsentiert den aktuellen Zustand der Weiche und kann die Werte `STRAIGHT`, `RIGHT`, `LEFT` und `UNDEFINED` annehmen. `UNDEFINED` drückt aus, dass die jeweilige Weiche noch nicht gestellt wurde. Dies ist der Initialzustand einer Weiche. Ist nun eine Bedingung zum Umschalten der Weiche erfüllt, nimmt diese den entsprechenden Zustand an. Ansonsten behält die Weiche ihren vorherigen Zustand bei.
Die Variable `point_conflict` gibt an, ob für eine Weiche unterschiedliche Weichenstellungen angefordert wurden. Dies dient dazu, die Markierung von Weichenkonflikten im Route Conflict Table zu überprüfen.

```
MODULE point(straight_cond, left_cond, right_cond)

VAR
state : {UNDEFINED, STRAIGHT, RIGHT, LEFT};
point_conflict : boolean;

ASSIGN
init(state) := UNDEFINED;
init(point_conflict) := 0;

next(state) := case
       straight_cond : STRAIGHT;
```

```
       left_cond : LEFT;
       right_cond : RIGHT;
       1 : state;
       esac;

next(point_conflict) := case
      straight_cond & left_cond |
      straight_cond & right_cond |
      left_cond & right_cond : 1;
      1 : point_conflict;
      esac;
```

7.6.6.2.2 Initialisierung der Variablen Für alle in einem Gleisnetz enthaltenen aktiven Weichen werden Variablen erstellt. Dazu wird jeweils das Modul `point` aufgerufen und als Parameter werden die Bedingungen übergeben, die angeben, wann die Weiche den jeweiligen Zustand annehmen soll. Dabei sind nur Anforderungen möglich, die mit dem jeweiligen Weichentyp vereinbar sind.

```
w1022 : point(w1022_straight, w1022_left, none);
w1018 : point(w1018_straight, none, w1018_right);
w1020 : point(w1020_straight, none, w1020_right);
```

7.6.6.2.3 Bedingungen zum Stellen der Weichen Eine Weiche soll in eine bestimmte Richtung gestellt werden, wenn eine Route befahren werden soll, für die im Point Position Table die entsprechende Weichenstellung definiert wurde.
So soll in dem Beispiel die Weiche `w1022` auf geradeaus gestellt werden, wenn die Route `route25` befahren werden soll und nach links, wenn die Route `route26` befahren werden soll.

```
w1022_straight := case
      route_A = route25 | route_B = route25 : 1;
      1 : 0;
      esac;

w1022_left := case
      route_A = route26 | route_B = route26 : 1;
      1 : 0;
      esac;

w1018_straight := case
      route_A = route23 | route_B = route23 : 1;
      1 : 0;
      esac;

w1018_right := case
      route_A = route24 | route_B = route24 : 1;
```

```
      1 : 0;
      esac;

w1020_straight := case
      route_A = route27 | route_B = route27 : 1;
      1 : 0;
      esac;

w1020_right := case
      route_A = route28 | route_B = route28 : 1;
      1 : 0;
      esac;
```

7.6.6.3 Signalverhalten

Nun wird die Modellierung des Signalverhaltens dargestellt.

7.6.6.3.1 Allgemeines Signalverhalten In dem Modul `signal` wird das Verhalten beschrieben, das für alle Signale gleich ist. Als Parameter werden drei Bedingungen übergeben, die angeben, ob eine Anforderung existiert, das entsprechende Signal geradeaus, nach links oder nach rechts zu stellen.
Die Variable `state` repräsentiert den aktuellen Zustand des Signals und kann die Werte `STOP`, `STRAIGHT`, `RIGHT` und `LEFT` annehmen. Der Initialzustand eines Signals ist `STOP`. Ist nun eine Bedingung zum Umschalten des Signals erfüllt, nimmt dieses den entsprechenden Zustand an. Ansonsten behält das Signal seinen vorherigen Zustand bei.
Bei der Verschlusstabellenüberprüfung ist es nicht notwendig, ein Signal wieder zurück auf `STOP` zu stellen. Wann ein Signal auf `STOP` gestellt werden soll, hat unmittelbar nichts mit den in den Verschlusstabellen gemachten Angaben zu tun, sondern mit der Spezifikation des Systems.

```
MODULE signal(straight_cond, left_cond, right_cond)

VAR
state : {STOP, STRAIGHT, RIGHT, LEFT};

ASSIGN
init(state) := STOP;

next(state) := case
      straight_cond : STRAIGHT;
      left_cond : LEFT;
      right_cond : RIGHT;
      1 : state;
      esac;
```

7.6.6.3.2 Initialisierung der Variablen Auch für alle in einem Gleisnetz enthaltenen Signale werden Variablen erstellt. Dazu wird jeweils das Modul `signal` aufgerufen und als Parameter werden die Bedingungen übergeben, die angeben, wann das Signal den jeweiligen Zustand annehmen soll. Dabei sind nur Anforderungen möglich, die mit dem jeweiligen Signaltyp vereinbar sind.

```
s3014 : signal(s3014_straight, s3014_left, none);
s3013 : signal(s3013_straight, none, s3013_right);
s3015 : signal(s3015_straight, none, s3015_right);
```

7.6.6.3.3 Bedingungen zum Stellen der Signale Ein Signal soll in eine bestimmte Richtung gestellt werden, wenn eine Route befahren werden soll, für die im Signal Setting Table die entsprechende Signalstellung definiert wurde.
Das Signal `s3014` soll in dem Beispiel auf geradeaus gestellt werden, wenn die Route `route25` befahren werden soll, und nach links, wenn die Route `route26` befahren werden soll.

```
s3014_straight := case
      route_A = route25 | route_B = route25 : 1;
      1 : 0;
      esac;

s3014_left := case
      route_A = route26 | route_B = route26 : 1;
      1 : 0;
      esac;

s3013_straight := case
      route_A = route23 | route_B = route23 : 1;
      1 : 0;
      esac;

s3013_right := case
      route_A = route24 | route_B = route24 : 1;
      1 : 0;
      esac;

s3015_straight := case
      route_A = route27 | route_B = route27 : 1;
      1 : 0;
      esac;

s3015_right := case
      route_A = route28 | route_B = route28 : 1;
      1 : 0;
      esac;
```

7.6.6.4 Sensorverhalten

Schließlich folgt noch die Modellierung das Sensorverhaltens.

7.6.6.4.1 Allgemeines Sensorverhalten In dem Modul `sensor` wird das Verhalten beschrieben, das für alle Sensoren gleich ist. Als Parameter wird eine Bedingungen übergeben, die angibt, wann der Sensor erreicht werden kann.
Die Variable `reached` gibt an, ob ein Sensor bereits erreicht wurde. Wurde der Sensor noch nicht erreicht und ist die als Parameter übergebene Bedingung `cond` erfüllt, wird er im nächsten Schritt erreicht.
Im Gegensatz zur Prüfung des Systemmodells sind hier keine Sensorzähler notwendig, da es zur Prüfung der Verschlusstabellen ausreicht, wenn sich eine Bahn auf einer Route befindet.

```
MODULE sensor(cond)

VAR
reached : boolean;

ASSIGN
init(reached) := 0;

next(reached) := case
      cond : 1;
      1 : reached;
      esac;
```

7.6.6.4.2 Initialisierung der Variablen Für alle in einem Gleisnetz enthaltenen Sensoren werden Variablen erstellt. Dazu wird jeweils das Modul `sensor` aufgerufen und als Parameter die Bedingung übergeben, die angibt, wann der jeweilige Sensor erreichbar ist.

```
g4023 : sensor(g4023_cond);
g4025 : sensor(g4025_cond);
g4027 : sensor(g4027_cond);
g4125 : sensor(g4125_cond);
g4126 : sensor(g4126_cond);
g4127 : sensor(g4127_cond);
g4128 : sensor(g4128_cond);
g4129 : sensor(g4129_cond);
g4130 : sensor(g4130_cond);
```

7.6.6.4.3 Erreichbarkeitsbedingungen Ein Sensor kann erreicht werden, wenn ein anderer Sensor, von dem der jeweilige Sensor erreichbar ist, bereits erreicht wurde und die Signale und Weichen entsprechend gestellt sind. Eine Ausnahme stellen Route

Request Sensoren dar. Diese werden erreicht, wenn eine Route befahren werden soll, die den jeweiligen Sensor als Route Request Sensor hat.
So ist in dem Beispiel `g4023` der Route Request Sensor der Routen `route23` und `route24`. Wenn eine dieser Routen befahren werden soll, wird `g4023` demzufolge erreicht.

```
g4023_cond := case
      route_A = route23 | route_B = route23 : 1;
      route_A = route24 | route_B = route24 : 1;
      1 : 0;
      esac;

g4025_cond := case
      route_A = route25 | route_B = route25 : 1;
      route_A = route26 | route_B = route26 : 1;
      1 : 0;
      esac;

g4027_cond := case
      route_A = route27 | route_B = route27 : 1;
      route_A = route28 | route_B = route28 : 1;
      1 : 0;
      esac;
```

Wenn eine Bahn von einem Sensor auf einen Nachbarsensor zufährt und sich zwischen den beiden Sensoren ein Signal befindet, so wird der Nachbarsensor nur erreicht, wenn das Signal nicht im Zustand `STOP` ist. In dem Beispiel wird der Sensor `g4125` erreicht, wenn Sensor `g4023` erreicht wurde und das Signal `s3013` nicht auf `STOP` steht.
Häufig ist ein Sensor von einem anderen Sensor nur erreichbar, wenn bestimmte Weichenstellungen vorliegen. In dem Beispielgleisnetz ist der Sensor `g4126` z.B. ereichbar, wenn der Sensor `g4125` erreicht wurde und die Weiche `w1018` nach rechts gestellt ist oder wenn der Sensor `g4129` erreicht wurde und die Weiche `w1022` nach links gestellt ist.

```
g4125_cond := case
      s3013.state != STOP & g4023.reached : 1;
      1 : 0;
      esac;

g4126_cond := case
      w1018.state = RIGHT & g4125.reached : 1;
      w1022.state = LEFT & g4129.reached : 1;
      1 : 0;
      esac;

g4127_cond := case
```

```
      s3015.state != STOP & g4027.reached : 1;
      1 : 0;
      esac;

g4128_cond := case
      w1018.state = STRAIGHT & g4125.reached : 1;
      w1020.state = RIGHT & g4127.reached : 1;
      1 : 0;
      esac;

g4129_cond := case
      s3014.state != STOP & g4025.reached : 1;
      1 : 0;
      esac;

g4130_cond := case
      w1020.state = STRAIGHT & g4127.reached : 1;
      w1022.state = STRAIGHT & g4129.reached : 1;
      1 : 0;
      esac;
```

7.6.6.5 Verschlusstabellen

7.6.6.5.1 conflicts-Block Zunächst soll der conflicts-Block überprüft werden. In diesem sind bekanntlich die Routenkonflikte definiert, die zu Kollisionen führen können. Die Bedingung `safe1` beinhaltet die drei folgenden allgemeinen Sicherheitsbedingungen, die gelten müssen, damit keine Kollisionen auftreten.

```
safe1 := sr1 & sr2 & sr3;
```

- `sr1` enthält alle Sicherheitsbedingungen, die eingehalten werden müssen, damit keine Bahnen aus unterschiedlichen Richtungen auf dieselbe Kreuzung zufahren, so dass es zu einem Flankenzusammenstoß kommen kann.
- `sr2` enthält alle Sicherheitsbedingungen, die eingehalten werden müssen, damit keine Bahnen aus unterschiedlichen Richtungen auf dieselbe Weiche zufahren, so dass es zu einem Flankenzusammenstoß kommen kann.
- `sr3` beinhaltet alle Sicherheitsbedingungen, die eingehalten werden müssen, damit keine Bahnen in entgegengesetzter Fahrtrichtung aufeinander zufahren, so dass es zu einem Frontalzusammenstoß kommen kann.

Diese drei Bedingungen beinhalten wiederum die Sicherheitsbedingungen der jeweiligen Kategorie, die für das jeweilige Gleisnetz gelten müssen.
Die Sicherheitsbedingungen sind schon aus Abschnitt 7.6.4.7 auf Seite 228 bekannt. Der einzige Unterschied besteht darin, dass hier keine Sensorzähler verwendet werden, sondern nur geprüft wird, ob bestimmte Sensoren bereits erreicht wurden.

```
sr1 := sr1_1 & sr1_2 & sr1_3;

sr1_1 := !(g4125.reached & w1018.state = STRAIGHT &
           w1022.state = LEFT & g4129.reached);

sr1_2 := !(g4125.reached & w1018.state = STRAIGHT &
           w1020.state = STRAIGHT & g4127.reached);

sr1_3 := !(g4129.reached & w1022.state = LEFT &
           w1020.state = STRAIGHT & g4127.reached);

sr2 := sr2_1 & sr2_2 & sr2_3;

sr2_1 := !(g4129.reached & w1022.state = LEFT &
           w1018.state = RIGHT & g4125.reached);

sr2_2 := !(g4125.reached & w1018.state = STRAIGHT &
           w1020.state = RIGHT & g4127.reached);

sr2_3 := !(g4127.reached & w1020.state = STRAIGHT &
           w1022.state = STRAIGHT & g4129.reached);

sr3 := 1;
```

Jetzt wird angegeben, bei welchen Routenkombinationen im conflicts-Block ein Konflikt definiert ist. Für das Beispiel sieht dies folgendermaßen aus:

```
conf := case
      route_A = route23 & route_B = route26 : 1;
      route_A = route23 & route_B = route27 : 1;
      route_A = route23 & route_B = route28 : 1;
      route_A = route24 & route_B = route26 : 1;
      route_A = route25 & route_B = route27 : 1;
      route_A = route26 & route_B = route23 : 1;
      route_A = route26 & route_B = route24 : 1;
      route_A = route26 & route_B = route27 : 1;
      route_A = route27 & route_B = route23 : 1;
      route_A = route27 & route_B = route25 : 1;
      route_A = route27 & route_B = route26 : 1;
      route_A = route28 & route_B = route23 : 1;
      1 : 0;
      esac;
```

Die folgenden beiden Bedingungen drücken aus, dass, wenn für zwei Routen ein Konflikt markiert wurde, irgendwo eine Kollison möglich sein muss bzw. wenn kein Konflikt markiert wurde, keine Kollision möglich sein darf.

```
SPEC conf = 1 -> !(AG safe1)
SPEC conf = 0 -> AG safe1
```

7.6.6.5.2 point-conflicts-Block Nun soll der point-conflicts-Block überprüft werden, in dem die Weichenkonflikte definiert sind. Dazu wird eine Bedingung `safe2` definiert, die wahr ist, wenn keine Weichenkonflikte auftreten.

```
safe2 := !w1022.point_conflict &
         !w1018.point_conflict &
         !w1020.point_conflict;
```

Es wird angegeben, bei welchen Routenkombinationen im point-conflicts-Block ein Weichenkonflikt definiert ist.

```
point_conf := case
      route_A = route23 & route_B = route24 : 1;
      route_A = route24 & route_B = route23 : 1;
      route_A = route25 & route_B = route26 : 1;
      route_A = route26 & route_B = route25 : 1;
      route_A = route27 & route_B = route28 : 1;
      route_A = route28 & route_B = route27 : 1;
      1 : 0;
      esac;
```

Wenn ein Weichenkonflikt definiert wurde, muss auch einer auftreten bzw. wenn kein Weichenkonflikt definiert wurde, darf auch keiner existieren.

```
SPEC point_conf = 1 -> !(AG safe2)
SPEC point_conf = 0 -> AG safe2
```

7.6.6.5.3 routedefinitions-Block, conditions-Block und clearances-Block Es soll überprüft werden, ob die Routen so durchfahren werden, wie sie definiert wurden. Dazu braucht immer nur eine Route befahren zu werden, da Routenkonflikte hier keine Rolle spielen.
Wenn eine Route in der beabsichtigten Weise befahren wird, dann muss es immer zu einem Zustand kommen, in dem Ein- und Ausgangssensor der Route erreicht wurden und es muss immer zu einem Zustand kommen, in dem der Eingangssensor bereits erreicht wurde und der Ausgangssensor noch nicht.
Wenn in dem Beispiel die Route `route23` befahren wird, dann muss es in jedem Fall zu einem Zustand kommen, in dem der Eingangssensor `g4125` bereits erreicht wurde und der Ausgangssensor `g4128` noch nicht. Dadurch wird sichergestellt, dass zuerst der Eingangssensor und erst danach der Ausgangssensor erreicht wird. Zudem muss in jedem Fall ein Zustand erreicht werden, in dem sowohl Eingangssensor `g4125` als auch Ausgangssensor `g4128` erreicht wurden.

```
SPEC (route_A = route23 & route_B = no ->
      AF (g4125.reached & !g4128.reached)&
```

```
      AF (g4125.reached & g4128.reached))
SPEC (route_A = route24 & route_B = no ->
      AF (g4125.reached & !g4126.reached)&
      AF (g4125.reached & g4126.reached))
SPEC (route_A = route25 & route_B = no ->
      AF (g4129.reached & !g4130.reached)&
      AF (g4129.reached & g4130.reached))
SPEC (route_A = route26 & route_B = no ->
      AF (g4129.reached & !g4126.reached)&
      AF (g4129.reached & g4126.reached))
SPEC (route_A = route27 & route_B = no ->
      AF (g4127._reached & !g4130.reached)&
      AF (g4127.reached & g4130.reached))
SPEC (route_A = route28 & route_B = no ->
      AF (g4127.reached & !g4128.reached)&
      AF (g4127.reached & g4128.reached))
```

Schließlich soll noch die Übereinstimmung der in clearances-Block und conditions-Block enthaltenen Angaben überprüft werden.
Das Signal `s3013` befindet sich im Beispielgleisnetz direkt an der spitzen Seite der Weiche `w1018`. Wenn das Signal nun einer Bahn die Fahrt in eine bestimmte Richtung erlaubt, dann muss `w1018` ebenfalls in die entsprechende Richtung gestellt sein.

```
no_signal_point_contradiction :=
      (s3013.state != STOP -> s3013.state = w1018.state) &
      (s3015.state != STOP -> s3015.state = w1020.state) &
      (s3014.state != STOP -> s3014.state = w1022.state);

SPEC AG no_signal_point_contradiction
```

Ruben Rothaupt, Taffou Happi

7.6.7 Reflexion

Am Anfang des Projektes wurde vorgegeben, dass mit Model Checking Techniken überprüft werden soll, ob die Projektierungsdaten eine sichere Steuerung des jeweiligen Gleisnetzes ermöglichen. Zu diesem Zweck wurde die Model Checking-Gruppe gegründet.
Da die Projektierungsdaten gleisnetzspezifisch sind, müssen sie für jedes zu steuernde Gleisnetz neu überprüft werden, was automatisiert werden soll. Das endgültige Programm ist in der Lage, festzustellen, ob alle notwendigen Sicherheitsbedingungen erfüllt sind. Somit ist das Ziel der Gruppe erreicht worden.
Um dieses Ziel zu erreichen, waren unterschiedliche Arbeitsschritte notwendig, die im Folgenden beschrieben werden sollen. Zudem sollen die Probleme betrachtet werden, die bei der Realisierung des Ziels aufgetaucht sind.

Der Begriff Model Checking war zunächst für alle Mitglieder der Gruppe neu. Um in das Thema einzusteigen, wurde das Seminar „Methoden der Verifikation“ der „Arbeitsgruppe Rechnerarchitektur“ empfohlen. In diesem Seminar hielten wir einen Vortrag zum Thema „CTL Model Checking“. Bis dahin war der Begriff Model Checking immer sehr abstrakt geblieben.
Nach mehreren Treffen mit Prof. Rolf Drechsler von der „Arbeitsgruppe Rechnerarchitektur“ erfuhren wir u.a., dass es für uns geeignete Model Checker Tools gibt, mit denen das System spezifiziert werden kann, um zu überprüfen, ob in dem System bestimmte Eigenschaften gelten.
Es wurde in der Gruppe beschlossen, den Model Checker NuSMV zu verwenden, da dieser unsere Anforderungen erfüllt. Dieser Model Checker wurde u.a. auch von Prof. Rolf Drechsler empfohlen.
Das erste Arbeitspaket war die Ableitung der Sicherheitsbedingungen, die im System erfüllt sein müssen. In [HP03a] wurde schon eine Liste von Sicherheitsbedingungen angegeben. Allerdings musste noch überprüft werden, ob diese korrekt und vollständig sind. Dazu wurde eine Fehlerbaumanalyse durchgeführt.
Es musste ein Modell des Systems in Form eines Transitionssystems in der Eingabesprache des Model Checkers erstellt werden. Das Modell besteht bekanntlich aus einem physikalischen Modell des betrachteten Gleisnetzes und einem Modell des Controllers.
Eine Beschreibung, wie ein physikalisches Modell aussehen kann, wurde in [HP03a] gegeben. Anhand dieser Beschreibung konnten bereits Konzepte zur Erstellung des physikalischen Modells entwickelt werden.
Allerdings blieb zunächst relativ unklar, wie das Modell des Controllers im Detail aussehen sollte. Mit Hilfe des Papers [PGHD04], das erst im Dezember 2004 zur Verfügung stand, wurde das Prinzip des Controller Modells deutlicher. Auch stand erst später eine Spezifikation des Steuerungssystems zur Verfügung. So konnte zu diesem Zeitpunkt noch kein sinnvolles Controller Modell erstellt werden, da das Verhalten des Steuerungssystems noch nicht spezifiziert war. Da auch noch keine Projektierungsdaten verfügbar waren, war eine automatische Generierung des Controller Modells ohnehin noch nicht möglich. Zudem existierten noch keine genauen Angaben darüber, welche Informationen die Projektierungsdaten enthalten sollten.
Daher haben wir zunächst eine Überprüfung der Verschlusstabellen durchgeführt. Dies war möglich, da für die Verschlusstabellenüberprüfung keine Projektierungsdaten und keine Spezifikation des Steuerungssystems benötigt werden.
Da das System automatisch verifiziert werden sollte, mussten für jedes Gleisnetz die notwendigen Sicherheitsbedingungen automatisch abgeleitet werden und das Transitionssystem musste automatisch erstellt werden. Dazu wurden Informationen aus der TND und den Projektierungsdaten benötigt.
Um die in der TND enthaltenen Informationen zu verwenden, musste ein Parser zum Einlesen der TND implementiert werden. Es wurde im Plenum beschlossen, den Parser der Compiler Gruppe zu verwenden. Häufige TND-Änderungen führten zudem dazu,

dass auch unser Programm mehrere Male an die aktuelle Version angepasst werden musste.
Die Verifikation wurde für die vier Gleisnetze ausgeführt, die im Plenum als Teststrecke anerkannt wurden. Es ließ sich dabei nachweisen, dass alle Sicherheitsbedingungen erfüllt waren.
Schnittstellen bestanden vor allem zu der Compiler-Gruppe, da die von dieser Gruppe generierten Projektierungsdaten überprüft werden sollten. Zudem existierten Berührungspunkte mit der TND-Gruppe, da bestimmte Informationen aus der TND benötigt wurden und somit dort enthalten sein müssten. Außerdem gab es eine Abhängigkeit von der Steuerinterpreter-Gruppe, da diese für die Erstellung der Systemspezifikation zuständig war, die benötigt wurde, um das Controller Modell zu erstellen. Zudem wurde zur Erstellung des Entwicklungs- und Verifikationsprozesses mit der Steuerinterpreter-Gruppe und der Testgruppe zusammengearbeitet.
Am Anfang des dritten Semesters haben zwei Mitglieder der Gruppe das Projekt verlassen. Dieser Ausfall wurde von den restlichen Mitgliedern verkraftet.
Da das erstellte Programm automatisch überprüft, ob alle Sicherheitsbedingungen in dem Transitionssystem, das das Verhalten des Systems modelliert, erfüllt sind, sind wir mit unserem Ergebnis zufrieden.

Ruben Rothaupt, Taffou Happi

7.7 Hardware/Software-Test

Heutzutage wird immer mehr von verschiedenen Computersystemen abhängig, die die Kontrolle und Steuerung von vielen Tätigkeiten im Leben übernehmen. Das Spektrum reicht dabei von elektrischen Koch- und Backgeräten bis zu Steuerungssystemen, deren Störung finanzielle Schäden auslösen oder sogar Mensch und Umwelt gefährden können. Aus diesen Gründen ist das zuverlässige Funktionieren von Computersystemen besonders wichtig. Bevor diese zum Einsatz kommen, sollten sie auf korrekte Funktionsweise geprüft werden. Dies spart zum einen Kosten (falls ein Ausfall während des Betriebs auftreten sollte), verhindert aber auch katastrophale Folgen einer Havarie (z.B. Unfall eines Schnellzuges oder Atomkraftwerkes). Das betrifft genauso die Software wie die Hardware des gesamten Systems. Es wird also erst einmal geprüft, ob auf der Softwareseite keine Fehler vorkommen (d.h. ob die Programme korrekt funktionieren und die gewünschten Aufgaben erfüllen), dann werden die Funktionen der Software im Zusammenspiel mit der Hardware geprüft. Am Ende wird ein Integrationstest durchgeführt, wobei geprüft wird, ob die Software und Ziel-Hardware zusammenspielen. Wenn ja, so heißt dies, dass das Gesamt-System korrekt zusammen funktioniert. Andernfalls darf das System in seiner vorliegenden Form nicht zum Einsatz kommen.

Marcin Dysarz

7.7.1 Überblick

Zu jeder Entwicklung gehört auch ein entsprechender Testvorgang. Ein Test dient der Qualitätssicherung einer neu erstellten Software. Dabei geht es prinzipiell darum, das tatsächliche Verhalten mittels Testfällen zu untersuchen und die Ergebnisse mit den erwarteten Ergebnissen (Spezifikation) zu vergleichen und zu dokumentieren.

In diesem Kapitel werden die Testkonzepte und Testvorgänge beschrieben, die ein zuverlässiges Funktionieren des TRACS-Steuersystems gewährleisten. Das Konzept umfasst automatische Testgenerierung aus der Anfangsspezifikation sowie Testausführung und -auswertung mit Hilfe des RT-Testers (ein Testwerkzeug, das von der Firma Verified [Ver] entwickelt worden ist). Es ist ein Modell eines Ebenen-Tests gebaut worden, anhand dessen sind die Tests durchgeführt worden, soweit dazu Zeit vorhanden war. Weitere, nicht mehr durchgeführte Tests werden als Konzept beschrieben.

Marcin Dysarz

7.7.2 Testkonzepte

Als Wissensbasis für die Erstellung folgender Kapitel dienten uns die Papers [HP02], [Pel03], [HP03b], [Pel02], [PT02], [PZ99] und [Ver04], sowie das während der Vorlesungen Testautomatisierung 1 und 2 sowie Safety Critical Systems 1 erworbene Wissen.

7.7.2.1 Automatische Testfallgenerierung

Um Wiederverwendbarkeit der Testprozeduren für verschiedene Gleisnetze zu erreichen, müssen die jeweiligen Testfälle für ein konkretes Gleisnetz aus der Gleisnetzspezifikation abgeleitet werden (siehe auch Kapitel 4.3.2 auf Seite 43).

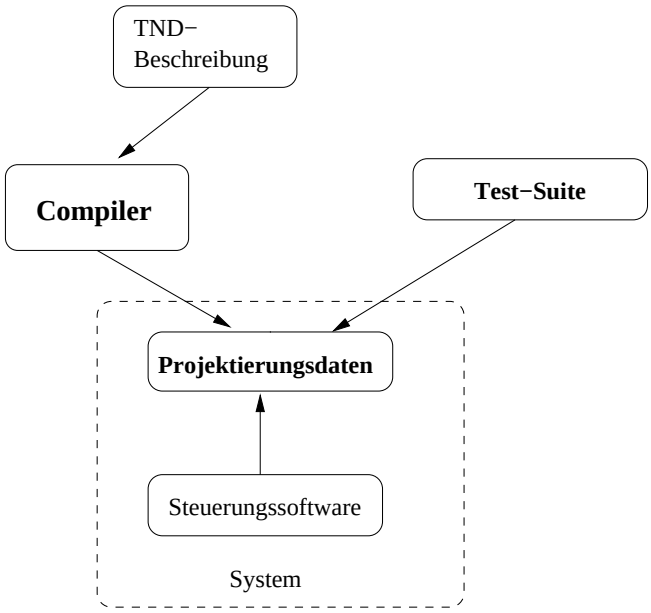


Abbildung 7.49: Projektierungsdaten, Generierung und Verwendung

Wie man in Abbildung 7.49 sieht, werden aus der TND-Beschreibung automatisch (mit Hilfe des Compilers) die Projektierungsdaten für die Steuerungssoftware generiert.

Diese sind auch als Ausgangs-Konfigurationsdatei für die Testprozeduren benutzt worden. Da alle für die Steuerung relevanten Gleisnetzinformationen in dieser Datei vorhanden sind, müssen diese Daten zwangsläufig auch ausreichen, das Steuerungssystem zu testen.

Anhand dieser Informationen und der Systemspezifikation wurden entsprechende Module (Testgenerator, Testorakel, Testchecker) im Testvorgang vorbereitet und benutzt.

Hierbei stellt sich allerdings auch die Frage der Korrektheit der Projektierungsdaten. Sind schon diese Projektierungsdaten fehlerhaft, so kann ein Test keine sinnvollen Ergebnisse über die Korrektheit der Software liefern.
Um die Testprozeduren ganz unabhängig vom System zu machen, sollte man einen separaten Compiler programmieren, der die Ausgangs-Konfigurationsdatei aus der TND-

Beschreibung erstellt. Aufgrund einer nicht ausreichenden Anzahl von Projektteilnehmern und des voraussichtlichen Aufwandes, der damit verbunden wäre, haben wir das nicht getan. Wir haben die Projektierungsdaten benutzt, die auch vom System benutzt werden und sind davon ausgegangen, dass diese korrekt waren, da die Korrektheit der Projektierungsdaten mit Hilfe von Model Checking Methoden gezeigt wird. Siehe hierzu 7.6 auf Seite 203.

Marcin Dysarz

7.7.2.2 Ebenen-Test

Ein vollständiger Test besteht aus mehreren Teil-Tests, die auf mehreren Ebenen vollgezogen werden. Heutzutage werden Tests üblicherweise auf folgenden Ebenen durchgeführt:

- Unit Test
- Software-Integrationstest
- Hardware/Software-Integrationstest
- System-Integrationstest

Wenn die Tests auf allen Ebenen mit einem positiven Ergebnis abgeschlossen werden, dann kann man behaupten, dass die Vollständigkeit der Tests garantiert ist und das System korrekt funktioniert.

Unit Tests (Komponententests) überprüfen einzelne Komponenten, z.B. eine einzelne Funktion innerhalb der Software. Dabei wird die Umgebung simuliert, d.h. die Umgebungsvariablen sowie die globalen Variablen werden voreingestellt. Zum Überprüfen der Funktionen verwendet man Test-Generator, Test-Orakel und Test-Checker. Als Schnittstelle dienen hier die Übergabeparameter und Rückgabewerte. Anhand dieser werden die Funktionen auf ihr korrektes Funktionieren geprüft. Durch parallele Ausführung und Vergleich der Ergebnisse zwischen der zu testenden Funktion und einem Test-Orakel wird eine Test-Entscheidung getroffen.

Software-Integrationstests überprüfen die sich gegenseitig beeinflussenden Software-Komponenten. Dabei werden das Ein- und Ausgabeverhalten des Systems sowie Software-Komponenten innerhalb des Systems getestet. Die Hardware (die zu steuernden Komponenten, Weichen, Signale, Sensoren) werden hierbei durch sogenannte Stubs simuliert. Zum Testen wird die Softwareschnittstelle zur Steuerungssoftware genutzt.

Hardware/Software-Integrationstests testen ein Sub-System, das aus Originalhardware und integrierter Software besteht (Steuerhard- und software mit I/O-Karten), in

Blackbox-Form. Dies bedeutet, von Interesse ist hier das Ein- und Ausgabeverhalten der Steuerungssoftware, nicht aber das interne Verhalten. Die Testumgebung ist hierbei eine Simulation der zu steuernden Komponenten (Gleisnetz) und die eigene Hardware zur Verarbeitung der Steuerimpulse. Als Schnittstelle dienen hier I/O-Karten, durch die der Testling und der Testrechner verbunden sind.

System-Integrationstests testen ein komplettes System innerhalb seiner operationalen Umgebung oder einer Simulation davon. System-Tests werden normalerweise als Blackbox-Tests gegen die Systemanforderungen durchgeführt. Schnittstelle für die Tests sind I/O-Karten, an denen aber die richtige Hardwareelemente (Weichen, Signale, Sensoren) angeschlossen werden.

Marcin Dysarz

7.7.3 Testumgebung

Um die TRACS Steuerungssoftware zu testen, war eine Testumgebung notwendig. Diese bestand aus einem Computer, der über die notwendige Testsoftware verfügte. Für Tests, die die Hardware mit einbeziehen, hätte man den Computer noch mit bestimmter Hardware (z.B. IO-Karten) ausstatten müssen. Zur Kommunikation zwischen Testumgebung und dem Testling (Softwaresystem) können dann, je nach Testebene, verschiedenene Kommunikationskanäle eingesetzt werden. Diese sind:

- Umgebungsvariablen, globale Variablen
- Übergabeparameter und Rückgabewerte
- Shared Memory
- I/O-Karten bzw. -Leitungen

Bei den von uns durchgeführten Unit-Tests wurde sich auf Übergabeparameter und Rückgabewerte beschränkt, da die hier zu testenden Funktionen in dieser Form alle von ihnen benötigten Daten bekommen können.
Durch die Kanäle fand der Datenaustausch zwischen beiden Systemen statt. Testeingaben wurden in den Testling eingespeist, und anhand dessen Ausgaben wurde das Testergebnis bestimmt.
Die Voraussetzung für dies war ein Rechner mit Betriebssystem, welcher die von der Testsoftware (und dem Testling, solange es um Software-Tests geht) benötigten Funktionalitäten anbot.

Für die TRACS Testumgebung war ein PC-Computer benötigt, auf dem ein Linux-Betriebssystem installiert war. Ein Linuxsystem ist notwendige Voraussetzung für den

RT Tester 6.0, der als Testsoftware für die TRACS Steuerungssoftware eingesetzt wurde. Die Distribution spielt eine sekundäre Rolle. Wir hatten drei Distributionen (Debian, Gentoo, SuSE) im Einsatz und keine Probleme bei der Testerstellung und -durchführung festgestellt.
Die Hardwareanforderungen an diesen Rechner sind nicht weiter bestimmt.

Auf Grund von Verspätungen bei der Bereitstellung des Systems und halbierter Anzahl der Teilnehmer (im Vergleich zum Anfang des Projektes) war nur Software-Test-Durchführung möglich. Zum Verlauf der Hardware-Tests wird daher nur ein Vorschlag gemacht.
Die erste, für das Testen der TRACS Steuerungssoftware notwendige Aufgabe bestand darin, eine solche Testumgebung aufzusetzen, so dass erforderliche Tests in die Testumgebung integriert und dort ausgeführt werden konnten. Dazu musste zunächst das Betriebssystem installiert werden. War auf dem Testrechner ein entsprechendes Betriebssystem eingerichtet, konnte der RT-Tester installiert werden. Die beiden Aufgaben bereiteten keine weiteren Probleme.

Um mit dem RT-Tester verschiedene Testfälle auszuführen und auswerten zu können, musste für das TRACS Steuersystem ein Testprojekt erstellt werden, in dem alle Tests für die Steuerungssoftware logisch zusammengefasst wurden. Im Testprojekt sind Angaben enthalten, die alle auszuführenden Tests gemein haben, also zum Beispiel Umgebungsvariablen und Pfadangaben der Testumgebung und des Testlings.

Marcin Dysarz

7.7.3.1 Verified's RT Tester

In diesem Kapitel beschreiben wir den RT-Tester, das Werkzeug, das wir zum Testen unseres Systems benutzt haben.

Es gibt verschiedene Typen von Test Spezifikationen. Ursprünglich war es so, dass für jeden Typ das entsprechende Test-Tool benutzt werden musste, z.B. für Unit Tests ein Werkzeug, das auf low-level Software Anforderungen basiert und für die anderen Tests wieder deren entsprechende Tools, so dass man mit vielen unterschiedlichen speziellen Tools arbeiten musste.

Der RT-Tester macht dies nun als ein einziges Tool für alle Typen von Tests. Die Abbildung 7.50 auf der nächsten Seite zeigt den ursprünglichen Testansatz, Abbildung 7.51 auf Seite 261 zeigt den Aufbau des RT Testers. Beide stammen aus [PZ99].

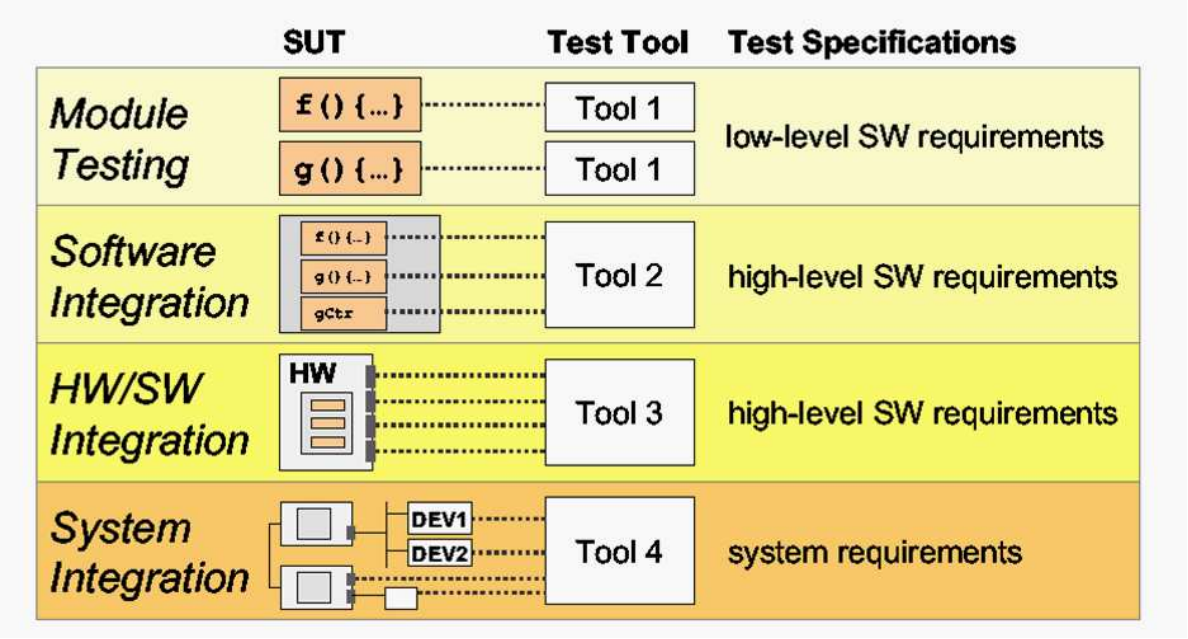


Abbildung 7.50: Ursprünglicher HW/SW Testansatz

Abstract Machine Layer
Der Abstract Machine Layer (AML) besteht aus den Abstract Machines (AMs), die vorkompilierte ausführbare Test-Spezifikationen in Echtzeit interpretieren. Dabei können AMs parallel zueinander Tests durchführen. AMs implementieren einen Algorithmus, um die Test-Daten automatisch zu erzeugen (Generator) und prüfen automatisch die Reaktion vom SUT (Checker). Die Schnittstellen werden als Channels bezeichnet. Diese Channels werden auf einer abstrakten Stufe kodiert und genutzt, die von den konkreten Software- oder Hardwareschnittstellen unabhängig ist, um ein konkretes SUT zu verbinden.

Interface Module Layer
Das Interface Module Layer (IFML) besteht aus den Interface Modules (IFMs). IFMs implementieren die Schnittstellenverfeinerung von der Schnittstelle der AMs zu der Schnittstelle des konkreten SUT. Die Abstraktion der Schnittstelle ist gerade umgekehrt, d.h sie wird von den Ausgaben des konkreten SUT zu den abstrakten AM Channels abgebildet.

Communication Control Layer
Das Communication Control Layer (CCL) überträgt die Ereignisse zwischen den AMs und den IFMs. Beispielsweise können verschiedene AMs Ausgaben generieren, die in das gleiche Channel c übertragen werden. Diese Ausgaben werden durch die Communication Control Layer zu den dazugehörigen IFMS abgebildet, und Channel c wird dann als ein

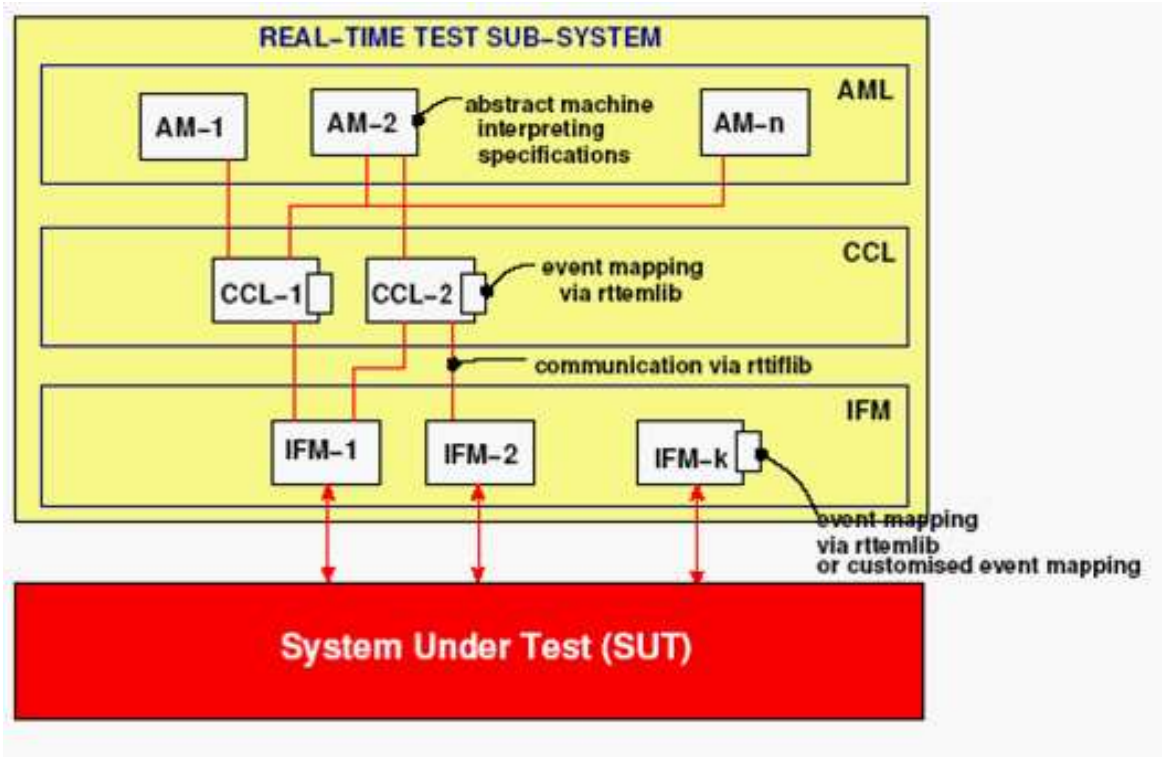


Abbildung 7.51: Aufbau des RT Testers

Input-Channel der AMs deklariert.

Testen mit dem RT Tester
Das organisatorische Modell, das oben beschrieben wird, wird im Folgenden berücksichtigt:

Das Testen von eingebetteten Systemen erfordert parallele Prozesse für die Simulation von externen Systemen in der SUT Betriebsumgebung und die Stimulation von spezifischen Ereignissen, um die Test-Durchführung zu kontrollieren und die SUT Reaktion auf Inputs, die von der Testumgebung gegeben werden, zu prüfen. Das einfache, sequenzielle Testen besteht aus zwei Schritten.
Als erstes sendet man Input Befehle zum SUT, dann überprüft man die Ausgabe des SUT und vergleicht diese mit der erwarteten Ausgabe. Solche Tests sind vor allem für Unit Test, aber nicht für viele komplexe Applikationen geeignet, bei denen einige logische Threads parallel zusammenarbeiten. In Situationen, bei denen das Verhalten des SUT abhängig ist von einigen Zustandsvariablen, die womöglich unabhängig voneinander verändert werden, funktioniert dies aufgrund der asynchronen Simulationen der verschiedenen Schnittstellen nicht. Ausgaben von einer AM werden oft nicht nur vom

SUT, sondern auch von den anderen AMs verarbeitet, deren kommendes Verhalten oder deren Testergebnisse abhängig von den Ereignissen der Ausgaben sind. Wenn eine SUT Schnittstelle verändert wird oder SWI Test (Software Integrationstest) Spezifikationen auf HSI Test (Hardware/Software Integrationstest) Stufe mehrmals benutzt werden, ist es praktisch, nur das zusammenführend Interface Module (IFM) gewechselt wird, ohne die Test Spezifikationen zu ändern.

Jun Liang und Jinghong Jin, bearbeitet von Marcin Dysarz

7.7.4 Testvorbereitung

Aus bereits genannten Gründen haben wir nur Software-Tests durchgeführt, d.h. Unit-Tests (die jeweilige Hauptfunktion der Software-Komponenten). Für die verschiedenen Ebenen der Integrationstests besteht ein Vorschlag.

Während der Testvorbereitungsphase konzentrierten sich die Aufgaben auf folgende Aspekte:

- Identifikation der Testziele
- Auswahl der Teststrecke
- Vorbereitung der Testfälle

Als die Grundlage für diese Aufgaben diente uns die im Kapitel 7.5 auf Seite 171 enthaltene Spezifikation des gesamten Systems.

7.7.4.1 Identifikation der Testziele

Für Software-Tests haben wir den Safety Controller (SC) genommen, der den Route Dispatcher (RD), die Route Controller (RC) und den Safety Monitor (SM) beinhaltet.

Für jede dieser Komponenten musste ein Test-Generator, ein Test-Orakel und ein Test-Checker vorbereitet werden.
Um die Testfälle aus der spezifischen Gleisnetzkonfiguration generieren zu können, musste außerdem eine Komponente zum Einlesen der Konfigurationsdatei (in diesem Fall Projektierungsdaten) erstellt werden. Diese musste jedoch nur einmal hergestellt werden, um dann von allen Test-Generatoren verwendet werden zu können.

Für den Implementierungvorgang haben sich somit folgende Aufgaben herausgestellt:

- Implementierung des Einlesens der Projektierungsdaten
- Implementierung der Testprozeduren für die zu testenden Komponenten

7.7.4.2 Auswahl der Teststrecke

Die Tests sollen möglichst originalgetreu die Realität eines Gleisnetzes abbilden. Dafür haben wir (mit Hilfe der BSAG) ein Beispielgleisnetz, mit vier Teststrecken, zusammengestellt.

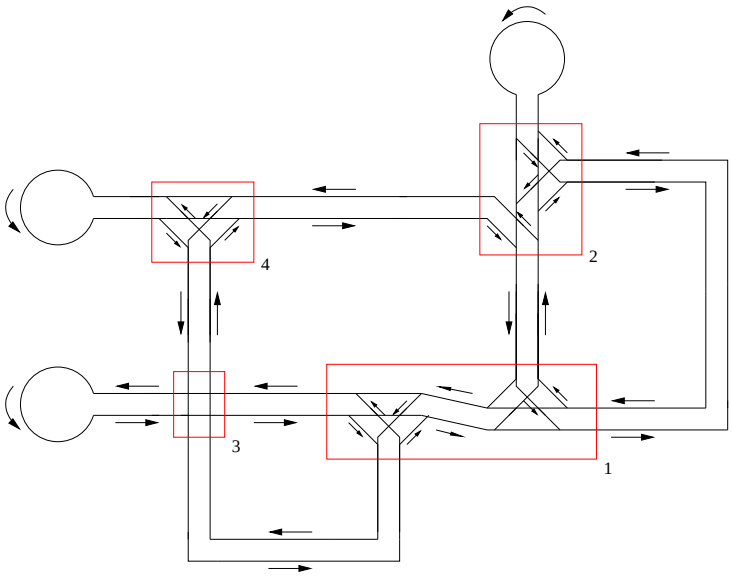


Abbildung 7.52: Gesamtes Testgleisnetz

Die Abschnitte, die sich in dem Testgleisnetz in roten Kästen befinden, sollen von einem Steuerungssystem verwaltet werden.

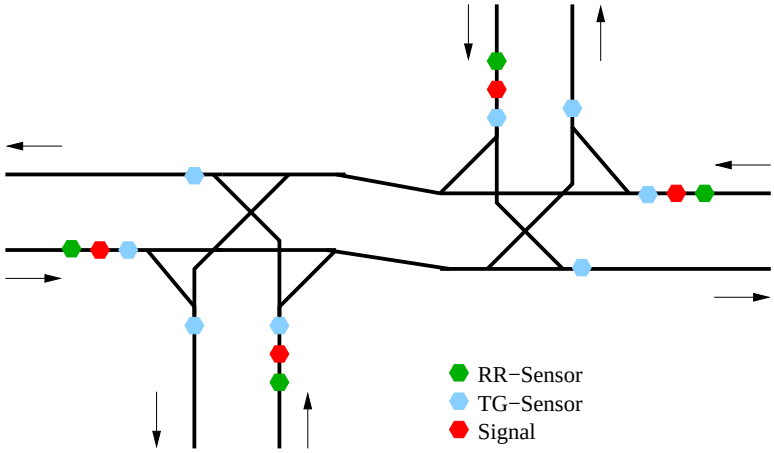


Abbildung 7.53: Teststrecke 1

Der komplizierteste Gleisabschnitt der Gesamtstrecke. Hohe Anzahl von Elementen, zudem viele verschiedene Typen. Hierbei handelt es sich um eine Abbildung der Domsheide (HB).

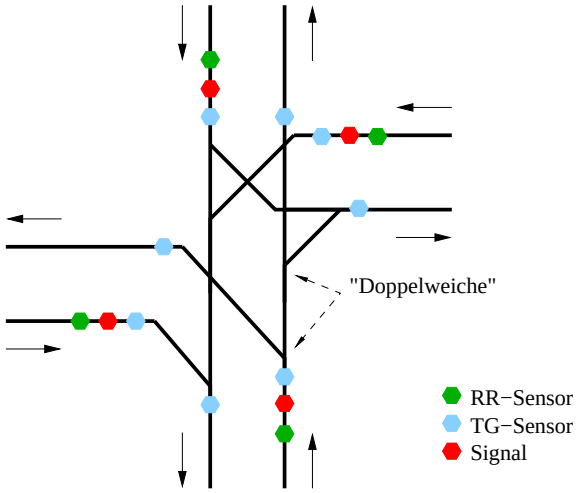


Abbildung 7.54: Teststrecke 2

Zweitkomplizierteste Strecke. In Richtung Süd-Nord ist eine Weiche vorhanden, die ein Auswahl links, rechts, geradeaus ermöglicht. Dies geschieht durch eine Zusammenkopplung von zwei einfachen, direkt hintereinander positionierten Weichen.

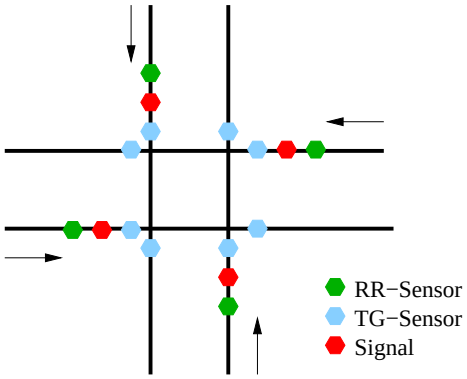


Abbildung 7.55: Teststrecke 3

Einfache Kreuzung, ohne Abbiegemöglichkeit.

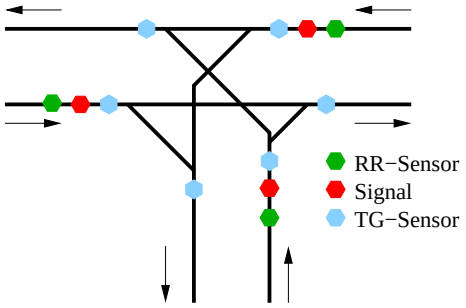


Abbildung 7.56: Teststrecke 4

Einfache Kreuzung, mit Links-Rechts-Abbiegesmöglichkeiten.

Auf diesen Teststrecken sind alle typischen Bauformen des Gleisnetzes dargestellt. Die Gleisnetzabschnitte 1 und 2 sind als Grundlage für die Unit-Tests benutzt worden. Nach der Beschreibung im TND-Format und Übersetzung durch den Compiler dienten diese als die Anfang-Konfigurationsdateien für den Testvorgang.

7.7.4.3 Vorbereitung der Testfälle

Für die RD, RC, SM ist ein Unit-Test durchgeführt worden (SM nicht komplett), für den gesamten SC war ein Software Integrationstest vorgesehen.

Unit-Tests (Komponententest) überprüfen die einzelnen Funktionen innerhalb der Steuerungssoftware. Die Tests spielen deshalb eine große Rolle in dem gesamten Testvorgang, weil man auf Grund der Feststellung, dass sämtliche Bauteile des Systems korrekt funktionieren, die Chancen erhöht, dass das Gesamtsystem auch korrekt funktioniert. Schnittstellen sind hier die globalen Variablen oder Übergabe-Parameter.

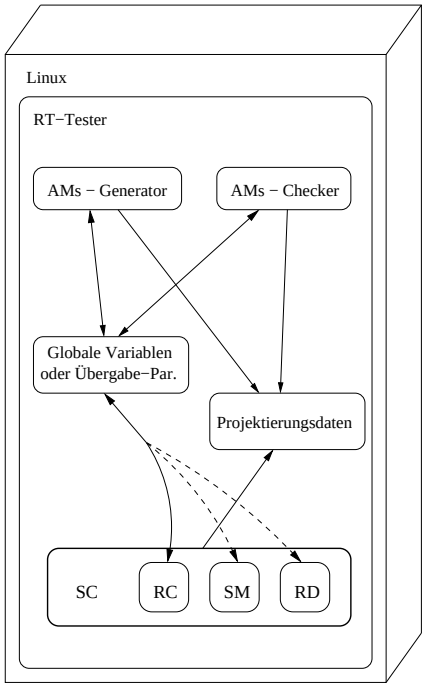


Abbildung 7.57: Schema der Unit-Tests

Folgende Komponenten sind zum Test notwendig:

- Steuerungssoftware mit Projektierungsdaten
- PC-Rechner, Betriebssystem, Testsoftware

Vorbereitung der Testfälle für die einzelnen Software-Komponenten ist ausführlich in dem Kapitel 7.7.5 auf der nächsten Seite beschrieben.

Software Integrationstests überprüfen sich gegenseitig beeinflussende Software-Komponenten in Form eines Blackbox-Tests. Die Tests überprüfen das gesamte Software-System. Schnittstelle ist hier Shared Memory (SHM).

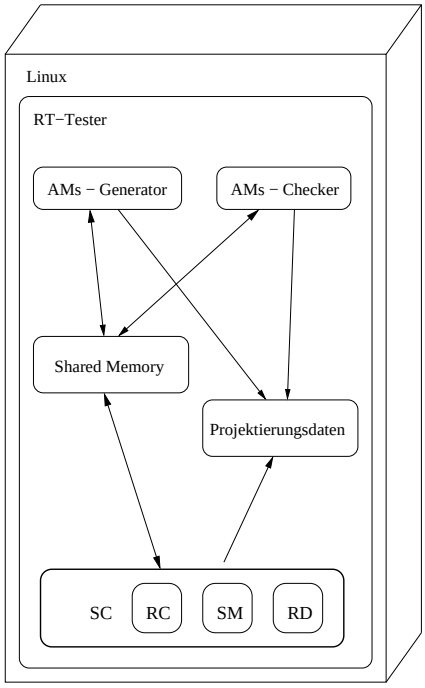


Abbildung 7.58: Schema der Software Integrationstests

Folgende Komponenten sind zum Test notwendig:

- Steuerungssoftware mit Projektierungsdaten
- PC-Rechner, Betriebssystem, Testsoftware

Ein Konzept für einen Software-Integrationstest ist ausführlich in dem Kapitel 7.7.7 auf Seite 303 beschrieben.

Marcin Dysarz

7.7.5 Test-Implementierung

Die wesentliche Arbeit bei der Implementierung der Tests bestand aus folgenden Arbeitspaketen:

- Implementierung des Einlesens der Projektierungsdaten
- Implementierung der Testprozeduren für den Route Dispatcher
- Implementierung der Testprozeduren für den Route Controller
- Implementierung der Testprozeduren für den Safety Monitor

Die Implementierung des Einlesens der Projektierungsdaten war dabei die Voraussetzung für die automatische Generierung der Testfälle.

Implementiert wurden für die zu testenden Komponenten jeweils ein Testgenerator, der auf Basis der Projektierungsdaten Kombinationen von Eingabedaten (nicht alle theoretisch möglichen – das würde in der Ausführung zu lange dauern – aber eine sinnvolle Untermenge davon) herstellt und diese einerseits an das SUT übergibt, andererseits auch an ein Testorakel. Das Orakel wurde gemäß der Spezifikation der zu testenden Komponente programmiert und soll die erwarteten Ausgaben produzieren. Im Checker letztlich werden nur noch Orakel-Output und SUT-Output verglichen. Aufgrund dieses Vergleiches kann die Beurteilung des Tests erfolgen. Für den Fall, dass Orakel-Output und SUT-Output inhaltlich gleich sind, gilt ein Test als bestanden, andernfalls als nicht bestanden.

Die genaue Beschreibung der einzelnen Testprozeduren erfolgt in den folgenden Kapiteln.

Marcin Dysarz, Arne Stahlbock

7.7.5.1 Einlesen der Projektierungsdaten

Zielsetzung dieses Arbeitspaketes war es, eine Repräsentation der als Binärdatei vorliegenden Projektierungsdaten im Speicher zu erhalten. Durch die SI-Gruppe war das Format dieser Datei vorgegeben (siehe 7.5.4.6 auf Seite 193), die hierbei verwendeten Datenstrukturen waren jedoch an einigen Stellen aus Testperspektive nicht immer günstig gewählt. Als Beispiel dafür lassen sich die Inline-Arrays anführen, deren Verwendung zur Folge hat, dass man, wenn man sich am Beginn eines solchen Arrays befindet, noch nicht weiß, wie lang dieser überhaupt ist. Benötigt man die Länge (was der SI möglicherweise nicht tut), hätte man die Elementanzahl durchzählen und dann wieder an den Anfang zurückspringen müssen. Um dies nicht immer wieder tun zu müssen, wurde beschlossen, für die Testumgebung eigene, vor allem um Längenangaben erweiterte, Datenstrukturen einzusetzen, in denen dann die Projektierungsdaten abgelegt werden sollen.
Da allerdings auch der Testling zum Betrieb mit Projektierungsdaten versorgt werden musste, und dieser natürlich „sein“ Format erwartete, mussten auch die von der SI-Gruppe vorgegebenen Strukturen benutzt werden. Der Einlesevorgang wurde daher so gestaltet, dass die aus der Datei extrahierten Daten gleichzeitig in zwei verschiedene Datenstrukturen im Speicher eingetragen wurden, einmal in die originalen SI-Projektierungsdatenstrukturen und einmal in die von uns erstellten erweiterten Strukturen, die dann von den Testprozeduren benutzt werden sollten.
Die erweiterten Strukturen werden nun dargestellt:

```
/* Offsets */
struct test_offsets_t {
    int version_offset;
    int rd_data_offset;
    int rc_data_offset;
    int sm_data_offset;
};

/* Versionsdaten */
struct test_version_t {
    int major;
    time_t build;
};

/* Struktur für RD */
struct test_rd_data_t {
    int max_routes;
    int max_queues;
    char *rct;
    char *queue_tab;
};

/* BEGIN Strukturen für RC */
struct test_shm_value_t {
    int id;
    uint16_t input;
    uint16_t input_mask;
    uint16_t output;
    uint16_t output_mask;
};

struct test_sensor_value_t {
    int id1;
    int id2;
    int relation;
    int value;
};

struct test_rc_data_t {
    int route_id;
    int num_point;
```

```
    struct test_shm_value_t *point_pos;
    int num_free_sig;
    struct test_shm_value_t *free_sig_pos;
    int num_lockfirst_sig;
    struct test_shm_value_t *lockfirst_sig_pos;
    int num_lock_sig;
    struct test_shm_value_t *lock_sig_pos;
    int num_req_sig;
    struct test_shm_value_t *req_sig_pos;
    int entry_sens_id;
    int exit_sens_id;
};
/* END Strukturen für RC */

/* BEGIN Strukturen für SM */
struct test_sm_condition_t {
    int type;
    void* condition;
    /*union {
        struct shm_value_t shm_value;
        struct sensor_value_t sensor_value;
    } condition;*/
};

struct test_sm_condition_list_t {
    int num_cond;
    struct test_sm_condition_t *sm_conditions;
};

struct test_hw_timeout_t {
    int id;
    time_t timeout;
};

struct test_sm_data_t {
    int num_cond_list;
    struct test_sm_condition_list_t *sm_condition_lists;
    int num_hw;
    struct test_hw_timeout_t *hw_timeouts;
};
/* END Strukturen für SM */
```

```
/* Hauptstruktur, enthält alle anderen */
struct test_projdata {
    struct test_offsets_t *offsets;
    struct test_version_t *version;
    struct test_rd_data_t *rd_data;
    int num_rc_data;
    struct test_rc_data_t *rc_data;
    struct test_sm_data_t *sm_data;
};
```

Beim Vergleich mit den SI-Datenstrukturen fällt auf, dass unsere Datenstrukturen diesen fast entsprechen. Zur Unterscheidung wurde das Namenspräfix `test` eingeführt, und alle eingeführten Längenangaben beginnen mit `num`.
Darüber hinaus ist die einzig nennenswerte Differenz, dass in
`struct test_sm_condition_t` statt der `union condition` ein `void *condition` eingeführt wurde, der dann, da ohnehin je nach Typ der condition immer nur eines der beiden union-Bestandteile vorliegen würde, auf eine Instanz eben dieses Bestandteils zeigen sollte.
Es wurden dann Funktionen geschrieben, die das Einlesen durchführen. Auf den Einsatz von Parsergeneratoren wurde an dieser Stelle von vorn herein verzichtet, da man insbesondere beim Erstellen einer Grammatik gewisse Probleme befürchtete. In einer Binärdatei sind letztlich ausschließlich einzelne Bytes nacheinander einzulesen, niemals Zeichenketten o.ä. – dabei haben allerdings die Bytes je nach ihrer Position und den Werten der vor ihnen stehenden Bytes andere Bedeutungen. Da letztlich die Implementierung ohne Parsergeneratoren auch nicht übermäßig lange gedauert hat, fühlen wir uns in dieser Wahl auch bestätigt.
Wir erklären nun einige (nicht alle) der zum Einlesen benutzten Funktionen.

```
int read_data(const char *filename, struct test_projdata *pdata,
  config_data_t *sidata) {

    int retval = 0;

    FILE *file;
    file = fopen(filename, "r");
    if (file == NULL) {
        retval = -1;
    }
```

```
    if (read_offsets(file, pdata, sidata) == -1) {
        retval = -1;
    }

    if (read_version(file, pdata, sidata) == -1) {
        retval = -1;
    }

    if (read_rd_data(file, pdata, sidata) == -1) {
        retval = -1;
    }

    if (read_rc_data(file, pdata, sidata) == -1) {
        retval = -1;
    }

    if (read_sm_data(file, pdata, sidata) == -1) {
        retval = -1;
    }

    return retval;
}
```

Hier handelt es sich um die Hauptfunktion, die dann später von außen aufgerufen werden sollte. Ihr übergeben wird der Dateiname der Projektierungsdaten-Datei sowie zwei Pointer auf die Datenstrukturen, in denen die Daten abgelegt werden sollten (wie beschrieben einmal im Originalformat, einmal in unserem erweiterten Format).
Diese Funktion ruft dann nacheinander Unterfunktionen auf, die die einzelnen Abschnitte bearbeiten sollen. Sobald irgendwo ein Fehler auftritt, wird -1 zurückgegeben, so dass die aufrufende Funktion feststellen kann, ob das Einlesen erfolgreich ist.
Als Beispiel für eine Unterfunktion wählen wir die folgende:

```
int read_rc_data(FILE *file, struct test_projdata *pdata,
  config_data_t *sidata) {

    int i;
    int j;
    long pos;
    int shm_count;
    int *intbuf = (int *) malloc(sizeof(int));
    uint16_t *uint16buf = (uint16_t *) malloc(sizeof(uint16_t));
```

```
// jump to rc_data_offset
if (fseek(file, pdata->offsets->rc_data_offset, SEEK_SET) != 0) {
    return -1;
}

if (pdata->rd_data->max_routes > 0) {

    // create structs according to number of routes
    pdata->num_rc_data = pdata->rd_data->max_routes;
    pdata->rc_data = (struct test_rc_data_t*)
      malloc(pdata->rd_data->max_routes *
      sizeof(struct test_rc_data_t));
    sidata->rc_data = malloc(pdata->rd_data->max_routes *
      sizeof(void*));
    for (i = 0; i < pdata->rd_data->max_routes; i++) {
        sidata->rc_data[i] = (struct rc_data_t *)
          malloc(sizeof(struct rc_data_t));
    }

    // iteration: number of routes
    for (i = 0; i < pdata->rd_data->max_routes; i++) {

        // read route ID
        if (read_int(file, intbuf) == 0) {
            //printf("R-ID: %d\n", *intbuf);
            pdata->rc_data[i].route_id = *intbuf;
            sidata->rc_data[i]->route_id = *intbuf;
        } else {
            return -1;
        }

        // remember position, count how many
// struct shm_value_t follow, then jump back
        pos = ftell(file);
        shm_count = count_shm_val(file);
        fseek(file, pos, SEEK_SET);

        // create structs according to number
        pdata->rc_data[i].num_point = shm_count;
        pdata->rc_data[i].point_pos = (struct test_shm_value_t*)
```

```
  malloc(shm_count * sizeof(struct test_shm_value_t));
sidata->rc_data[i]->point_pos = (struct shm_value_t*)
  malloc((shm_count+1) * sizeof(struct shm_value_t));

// first list are the point positions
for (j = 0; j < shm_count; j++) {
    if (read_int(file, intbuf) == 0) {
        //printf("Pt-ID: %d\n", *intbuf);
        pdata->rc_data[i].point_pos[j].id = *intbuf;
        sidata->rc_data[i]->point_pos[j].id = *intbuf;
    } else {
        return -1;
    }
    if (read_uint16(file, uint16buf) == 0) {
        //printf("Pt-Input: %d\n", *uint16buf);
        pdata->rc_data[i].point_pos[j].input =
          *uint16buf;
        sidata->rc_data[i]->point_pos[j].input =
          *uint16buf;
    } else {
        return -1;
    }
    if (read_uint16(file, uint16buf) == 0) {
        //printf("Pt-InputMask: %d\n", *uint16buf);
        pdata->rc_data[i].point_pos[j].input_mask =
          *uint16buf;
        sidata->rc_data[i]->point_pos[j].input_mask =
          *uint16buf;
    } else {
        return -1;
    }
    if (read_uint16(file, uint16buf) == 0) {
        //printf("Pt-Output: %d\n", *uint16buf);
        pdata->rc_data[i].point_pos[j].output =
          *uint16buf;
        sidata->rc_data[i]->point_pos[j].output =
          *uint16buf;
    } else {
        return -1;
    }
    if (read_uint16(file, uint16buf) == 0) {
```

```
            //printf("Pt-OutputMask: %d\n", *uint16buf);
            pdata->rc_data[i].point_pos[j].output_mask =
              *uint16buf;
            sidata->rc_data[i]->point_pos[j].output_mask =
              *uint16buf;
        } else {
            return -1;
        }
    }
    // terminate list in original structures
    sidata->rc_data[i]->point_pos[shm_count].id = -1;

    // jump back and count again
    // (make sure that we are at the end of the list)
    fseek(file, pos, SEEK_SET);
    shm_count = count_shm_val(file);

    // remember position, count how many
    // struct shm_value_t follow, then jump back
    pos = ftell(file);
    shm_count = count_shm_val(file);
    fseek(file, pos, SEEK_SET);

    // create structs according to number
    pdata->rc_data[i].num_free_sig = shm_count;
    pdata->rc_data[i].free_sig_pos = (struct test_shm_value_t*)
      malloc(shm_count * sizeof(struct test_shm_value_t));
    sidata->rc_data[i]->free_sig_pos = (struct shm_value_t*)
      malloc((shm_count+1) * sizeof(struct shm_value_t));

    // next are the "first signal" values
    for (j = 0; j < shm_count; j++) {
        if (read_int(file, intbuf) == 0) {
            //printf("FS-ID: %d\n", *intbuf);
            pdata->rc_data[i].free_sig_pos[j].id = *intbuf;
            sidata->rc_data[i]->free_sig_pos[j].id = *intbuf;
        } else {
            return -1;
        }
        if (read_uint16(file, uint16buf) == 0) {
            //printf("FS-Input: %d\n", *uint16buf);
```

```
                    pdata->rc_data[i].free_sig_pos[j].input =
                      *uint16buf;
                    sidata->rc_data[i]->free_sig_pos[j].input =
                      *uint16buf;
                } else {
                    return -1;
                }
                if (read_uint16(file, uint16buf) == 0) {
                    //printf("FS-InputMask: %d\n", *uint16buf);
                    pdata->rc_data[i].free_sig_pos[j].input_mask =
                      *uint16buf;
                    sidata->rc_data[i]->free_sig_pos[j].input_mask =
                      *uint16buf;
                } else {
                    return -1;
                }
                if (read_uint16(file, uint16buf) == 0) {
                    //printf("FS-Output: %d\n", *uint16buf);
                    pdata->rc_data[i].free_sig_pos[j].output =
                      *uint16buf;
                    sidata->rc_data[i]->free_sig_pos[j].output =
                      *uint16buf;
                } else {
                    return -1;
                }
                if (read_uint16(file, uint16buf) == 0) {
                    //printf("FS-OutputMask: %d\n", *uint16buf);
                    pdata->rc_data[i].free_sig_pos[j].output_mask =
                      *uint16buf;
                    sidata->rc_data[i]->free_sig_pos[j].output_mask =
                      *uint16buf;
                } else {
                    return -1;
                }
            }
            // terminate list
            sidata->rc_data[i]->free_sig_pos[shm_count].id = -1;

            [...]
}
```

An dieser Stelle brechen wir die Darstellung ab, da das System klar sein dürfte. Die in dieser Funktion verwendeten Hilfsfunktionen `read_int` und `read_uint16` tun nichts weiter, als ein Element des entsprechenden Typs aus der Datei zu lesen und in den übergebenen Puffer zu schreiben.
Letztlich füllen wir also zwei Datenstrukturen mit Inhalt, die dann in den Testprozeduren weiter verwertet werden.

Arne Stahlbock

7.7.5.2 Route Dispatcher

An dieser Stelle wird die Implementierung der Testprozeduren für den Route Dispatcher beschrieben. Dieser Vorgang erfolgte in mehreren Etappen:

- Bestimmung einer sinnvollen Menge von Testfällen anhand der RD-Spezifikation
- Implementierung der Funktionalität, aus den Projektierungsdaten diese Testfälle für das spezifische Gleisnetz zu erzeugen und sie in das SUT einzuspeisen (Testgenerator)
- Implementierung der Funktionalität, aus einem Testdatensatz den vom SUT erwarteten Output zu bestimmen (Testorakel)
- Implementierung der Funktionalität, den vom SUT gelieferten Output mit dem des Testorakels zu vergleichen (Testchecker)

7.7.5.2.1 Bestimmung einer Menge von Testfällen

Hierzu betrachten wir zunächst die Parameter der zu testenden Funktion:

```
void calculate_routes(shm_state_t *shm_shadow, struct rd_data_t *r,
  char *queue_tab, struct queue_t *queues, int *qpl, int *rst,
  int *bahn_counter, int *uebersprungene_queues)
```

Es werden also übergeben:

- der SHM-Zustand
- die Projektierungsdaten für den RD
- die Zuordnungstabelle von Routen zu Queues
- der aktuelle Zustand der Queues
- die Queue Priority List
- die Route State Table

- die Bahnzähler für jede Route
- der Zähler, wie oft die erste Queue übersprungen wurde

Vom SHM-Zustand werden lediglich die Route-Request-Sensoren betrachtet und da auch nur, ob sie ausgelöst haben - es muss also nicht jeder beliebige SHM-Zustand hergestellt werden. Im Gegenteil können wir hier die Zahl der Testfälle massiv verringern, indem wir pro Testfall entweder keinen oder nur einen RR-Sensor als ausgelöst setzen. Da das einzige, was mit ausgelösten RR-Sensoren geschehen soll, im Zurücksetzen des Sensors und Aufnahme der angeforderten Route in die zugehörige Queue ist, wobei Sensor für Sensor nacheinander abgearbeitet wird, kann man sich diesen Abschnitt des RD als Unterprozess vorstellen, der als Eingabe den Zustand der RR-Sensoren und der Queues hat. Beziehungsweise, da nacheinander gearbeitet wird, ist für jede einzelne Entscheidung nur der Zustand eines RR-Sensors und der ihm zugehörigen Queue relevant. Tests mit einer beliebigen Zahl gesetzter RR-Sensoren lassen sich prinzipiell daher alle auf solche mit nur einem oder keinem gesetzten RR-Sensor zurückführen bzw. würden diese nur wiederholen.
Die Projektierungsdaten und die Zuordnung von Routen zu Queues sind für ein Gleisnetz konstant, müssen also vom Testgenerator nicht variiert werden.
Die Queue-Zustände werden ebenfalls nicht völlig durchkombiniert, es wird nur das erste Element jeder Queue variiert. Die Begründung liegt darin, dass nur dieses erste Element relevant ist, wenn es darum geht, welche Route freigegeben wird. Nachfolgende Elemente in einer Queue haben nur dann eine Wirkung, wenn sie ein Request für die gleiche Route darstellen, die auch vom ersten Element verlangt wird. In dem Fall können ggf. gleich mehrere Requests erfüllt und der Bahnzähler dieser Route entsprechend erhöht werden. Jedoch erhält man solche Testfälle bereits dann, wenn das erste Element gesetzt ist und dann noch ein zweites über einen gesetzten RR-Sensor dazukommt. In Kombination mit der Variation der gesetzten RR-Sensoren reicht es also aus, nur das erste Queue-Element zu variieren, um letztlich alle möglichen Kombinationen der ersten zwei Elemente pro Queue zu erhalten. Auf das noch mögliche dritte wird dann allerdings verzichtet, um die Zahl der Testfälle nicht zu stark steigen zu lassen.
Die Queue Priority List besteht aus so vielen Elementen, wie Queues vorhanden sind (in der Regel nur wenige, die Höchstzahl bei den getesteten Gleisnetzen beträgt 4) und wird voll durchkombiniert.
Die möglichen Kombinationen in der Route State Table können ebenfalls verringert werden, wenn man sich vor Augen führt, dass für die Entscheidung, ob eine Route freigegeben werden kann, nur wichtig ist, ob eine in Konflikt dazu stehende Route aktiv ist oder nicht. Hier müssen für die Belegung der RST also nicht 4 mögliche Zustände pro Route (diese Anzahl definiert die SI-Spezifikation) berücksichtigt werden, sondern nur 2.
Die Bahnzähler müssen nicht variabel belegt werden. Da das einzige, was der RD mit ihnen machen soll, darin besteht, sie zu erhöhen, ihre Ausgangswerte aber nicht betrachtet

werden, können wir hier einen konstanten Wert als Testeingabe verwenden.
Schließlich fehlt noch der die Fairness herstellende Zähler, wie oft die erste Route übersprungen wurde. Hier muss zwischen zwei Werten variiert werden, da von dem Wert (bzw. dessen Erreichen einer Schwelle) eine Entscheidung abhängt; die beiden Testbelegungen sind folglich ein Wert unter und einer über der Schwelle.
Mit diesen Überlegungen haben wir die Zahl der Testfälle gegenüber dem wirklich kompletten Durchkombinieren aller möglichen Eingaben bereits deutlich eindämmen können, ohne dabei unserer Meinung nach relevante Testfälle auszulassen. Auch mit diesen Einschränkungen kann ein Testlauf bei einem großen Gleisnetz noch Stunden dauern, und wenn man betrachtet, wie schnell man sich auch nur eine Verdopplung der Testfälle einhandeln kann, von noch „schlimmeren“ Fällen nicht zu sprechen, dann bleibt auch nicht wirklich eine Wahl.

7.7.5.2.2 Testgenerator

Wir generieren die verschiedenen Testfälle, indem wir für jeden zu variierenden Eingangswert zwei Werte festhalten. Einer davon kann als `seed` bezeichnet werden, er charakterisiert den gerade für diese Eingangsvariable generierten Wert; der andere stellt das (konstante) Maximum für `seed` dar.
Es kann dann ein Code wie der folgende benutzt werden:

```
    // increase seed (go to next test case)
    gen_left_queues++;
    if(gen_left_queues == max_left_queues) {
        gen_left_queues = 0;
        gen_qpl++;
    }
    if(gen_qpl == max_qpl) {
        gen_qpl = 0;
        gen_rr++;
    }
    if(gen_rr == max_rr) {
        gen_rr = 0;
        gen_rst++;
    }
    if(gen_rst == max_rst) {
        gen_rst = 0;
        i = 0;
        gen_queues[i] += 1;
        while ((gen_queues[i] == max_queues[i]) &&
  (i < pdata->rd_data->max_queues)) {
            if (i == pdata->rd_data->max_queues - 1) {
```

```
                stop_tests = 1;
                i++;
            } else {
                gen_queues[i] = 0;
                gen_queues[i+1] += 1;
                i++;
            }
        }
    }
```

Es dürfte relativ leicht erkennbar sein, was hier geschieht: Es wird ein `gen_xyz` erhöht (entspricht dem angesprochenen seed), und falls dieser Wert sein Maximum erreicht hat, wird er auf Null zurückgesetzt und stattdessen ein anderer Wert erhöht. So erreicht man letztlich, dass jede Wertekombination im Laufe des Gesamtablaufs einmal vorliegt, und ganz am Ende wird eine Stop-Variable auf 1 gesetzt (um eine übergeordnete `while`Schleife zu verlassen).
Die Eingaben an Orakel und SUT werden dann einfach gesetzt mit:

```
    // set inputs
    reset_shm();
    set_tram_count();
    set_left_queues(gen_left_queues);
    set_qpl(gen_qpl);
    set_rst(gen_rst);
    set_rr(gen_rr);
    for (i = 0; i < pdata->rd_data->max_queues; i++) {
        set_queue(i, gen_queues[i], si_queues);
    }
```

Hier bekommen verschiedene `set`-Funktionen jeweils den zugehörigen `gen`-Wert übergeben. Sie nehmen dann die diesem Wert zugeordnete Umsetzung auf die tatsächlichen Eingabewerte vor.
An dieser Stelle dürfte es reichen, lediglich ein Beispiel für eine `set`-Funktion zu geben:

```
void set_rst(int seed) {

  int i;
  for (i = 0; i < pdata->rd_data->max_routes; i++) {
    rst[i] = ((seed / (pow(2, i))) % 2 == 0) ? RST_INAKTIV : RST_AKTIV;
    si_rst[i] = ((seed / (pow(2, i))) % 2 == 0) ?
      RST_INAKTIV : RST_AKTIV;
    if (debug) {
```

```
        printf("Setting state for route %d to %d\n", i, si_rst[i]);
    }
  }
}
```

`rst` und `si_rst` sind dabei die an Orakel bzw. SUT zu übergebenden Variablen, die hier belegt werden. Für die hier gezeigte Funktion ist die Umsetzung von `seed` auf die Belegung noch relativ simpel, es gibt da auch noch ganz andere Dinge... ;-)
Es muss nun noch gezeigt werden, wie man zu den textttmax-Werten gelangt, diese sind nämlich teilweise von dem vorliegenden Gleisnetz abhängig. So ist zum Beispiel die Anzahl der Routen im Netz bestimmend dafür, wie viele RST-Belegungen es geben kann, allein dadurch, dass die RST für jede Route ein Feld hat. Die Anzahl der Routen in einer Queue bestimmt die Anzahl der Belegungen für diese Queue usw. Das, sowie das Anlegen einiger Datenstrukturen (z.B. benötigen wir auch eine Zuordnung von Queue nach Route(n), während in den Projektierungsdaten nur Route nach Queue vorliegt) und das Setzen der `gen`-Werte auf den ersten Testfall geschieht hier:

```
// Calculate how many test cases we are facing.
// Set the gen_xyz values to first test case.
@printf("%d Queues found.", pdata->rd_data->max_queues);
gen_queues = (int *) malloc(sizeof(int) *
  pdata->rd_data->max_queues);
max_queues = (int *) malloc(sizeof(int) *
  pdata->rd_data->max_queues);
queue_info = (struct queue_data *) malloc(sizeof(struct queue_data)
  * pdata->rd_data->max_queues);
for (i = 0; i < pdata->rd_data->max_queues; i++) {
    gen_queues[i] = 0;
    count = 0;
    for (j = 0; j < pdata->rd_data->max_routes; j++) {
        if (pdata->rd_data->queue_tab[j] == i) {
            count++;
        }
    }
    @printf("Queue %d is responsible for %d routes.\n", i, count);
    queue_info[i].num_routes = count;
    queue_info[i].route_id = (int *) malloc(sizeof(int) * count);
    max_queues[i] = count + 1;
    @printf("It can therefore have %d different (test) states.\n",
      max_queues[i]);
    total *= max_queues[i];
    gen_queues[i] = 0;
```

```
        count = 0;
        for (j = 0; j < pdata->rd_data->max_routes; j++) {
            if (pdata->rd_data->queue_tab[j] == i) {
                queue_info[i].route_id[count] = j;
                count++;
            }
        }
    }
    max_rst = pow(2, pdata->rd_data->max_routes);
    @printf("There are %d routes.\nThe RST has %d (test) states.\n",
      pdata->rd_data->max_routes, max_rst);
    total *= max_rst;
    gen_rst = 0;
    max_qpl = fac(pdata->rd_data->max_queues);
    total *= max_qpl;
    @printf("The QPL can have %d states.\n", max_qpl);
    gen_qpl = 0;
    max_left_queues = 2;
    total *= max_left_queues;
    gen_left_queues = 0;
    max_rr = pdata->rd_data->max_routes + 1;
    total *= max_rr;
    gen_rr = 0;
    stop_tests = 0;

    @printf("Total: %lld test cases.\n", total);
```

7.7.5.2.3 Testorakel

Das Testorakel erhält die gleichen Eingaben wie das SUT und soll die vom SUT erwarteten Ausgaben bestimmen. (Es muss die Eingaben allerdings nicht in exakt gleicher Form bekommen, so kann z.B. hier auf eine Übergabe des kompletten SHM-Zustandes verzichtet und nur die Information zu den RR-Sensoren gegeben werden, während das SUT den gesamten SHM-Zustand verlangt, auch wenn es ihn nicht komplett auswertet. Die Eingaben müssen nur inhaltlich übereinstimmen und korrekte Vorhersagen ermöglichen.) Daher muss das Orakel gemäß der Spezifikation des SUT implementiert werden, ohne dabei allerdings das SUT zu kopieren. Das erreicht man am besten, indem man die SUT-Implementierung nicht betrachtet, bis das Orakel fertiggestellt ist. (Weicht man, wie oben beschrieben, bei den Eingaben von der Form, in der das SUT sie bekommt, ab, folgt daraus sogar zwangsläufig eine andere Implementierung.) Diesen Weg zu beschreiten ist auch im Wesentlichen gelungen, das SUT wurde erst bei den ers-

ten (fehlgeschlagenen) Testläufen richtig betrachtet, nur der Aufruf musste naturgemäß schon vorher bekannt sein.
Wir zeigen an dieser Stelle den Code, erläutern ihn aber nicht weiter. Was er tut, steht im Wesentlichen in der SI-Spezifikation in Kapitel 7.5.4.4 auf Seite 189, auch wenn von den in der Spezifikation geforderten Unterfunktionen hier abgewichen und alles in die eine Hauptfunktion gepackt wurde.

```
// This is the test oracle computing the expected outputs.
// It is implemented following the SI spec.
void rd_test_oracle() {

    int i;
    int j;
    int tmp;
    int curr_route;
    int confl_route;
    int has_conflict = 0;
    int stop = 0;

    if (rr < pdata->rd_data->max_routes) {
        if (queues[(int)(pdata->rd_data->queue_tab[rr])] == -1) {
            queues[(int)(pdata->rd_data->queue_tab[rr])] = rr;
            longer_queue = -1;
            longer_elem = -1;
        } else {
            longer_queue = pdata->rd_data->queue_tab[rr];
            longer_elem = rr;
        }
    } else {
        longer_queue = -1;
        longer_elem = -1;
    }
    for (i = 0; (i < pdata->rd_data->max_queues) && (stop == 0); i++) {
        curr_route = queues[qpl[i]];
        if (curr_route != -1) {
            has_conflict = 0;
            for (confl_route = pdata->rd_data->max_routes * curr_route;
              confl_route < pdata->rd_data->max_routes * (curr_route+1);
              confl_route++) {
                if ((pdata->rd_data->rct[confl_route] == 1) &&
                  (rst[confl_route % pdata->rd_data->max_routes] ==
```

```
                RST_AKTIV)) {
                    has_conflict = 1;
                }
            }
            if (has_conflict == 0) {
                rst[curr_route] = RST_AKTIV;
                tram_count[curr_route] += 1;
                if (longer_queue == qpl[i]) {
                    queues[qpl[i]] = longer_elem;
                    longer_queue = -1;
                    longer_elem = -1;
                } else {
                    queues[qpl[i]] = -1;
                }
                if (i != 0) {
                    *left_queues += 1;
                } else {
                    *left_queues = 0;
                }
                j = i;
                while (j < pdata->rd_data->max_queues - 1) {
                    tmp = qpl[j];
                    qpl[j] = qpl[j+1];
                    qpl[j+1] = tmp;
                    ++j;
                }
                --i;
            } else if ((*left_queues) >= pdata->rd_data->max_queues) {
                rst[curr_route] = RST_ANGEFORDERT;
                stop = 1;
            } else {
                rst[curr_route] = RST_ANGEFORDERT;
            }
        }
    }
}
```

7.7.5.2.4 Testchecker

Zum Schluss wird dann noch eine Funktion benötigt, die die von SUT und Orakel gelieferten Outputs vergleicht. Diese seiht in diesem Fall wie folgt aus:

```
// this is the checker comparing the SI outputs with the expected ones
int compare_results() {

    int retval = 1;
    int i;

    for (i = 0; i < pdata->rd_data->max_routes; i++) {
        if ((rst[i] == RST_INAKTIV) && (si_rst[i] != RST_INAKTIV)) {
            if (comp_debug) {
                printf("ERROR: SI has route %d at state %d while
                  checker has it at state %d\n",
                  i, si_rst[i], rst[i]);
            }
            retval = 0;
        } else if ((rst[i] == RST_ANGEFORDERT) && (si_rst[i] !=
          RST_ANGEFORDERT)) {
            if (comp_debug) {
                printf("ERROR: SI has route %d at state %d while
                  checker has it at state %d\n",
                  i, si_rst[i], rst[i]);
            }
            retval = 0;
        } else if ((rst[i] == RST_AKTIV) && (si_rst[i] != RST_AKTIV)) {
            if (comp_debug) {
                printf("ERROR: SI has route %d at state %d while
                  checker has it at state %d\n",
                  i, si_rst[i], rst[i]);
            }
            retval = 0;
        }
        if (tram_count[i] != si_tram_count[i]) {
            if (comp_debug) {
                printf("ERROR: SI has tram count for route %d at %d
                  while checker has it at %d\n",
                  i, si_tram_count[i], tram_count[i]);
            }
            retval = 0;
        }
    }
    if (*left_queues != *si_left_queues) {
        if (comp_debug) {
```

```
            printf("ERROR: SI has left_queues at %d while
              checker has it at %d\n",
              *si_left_queues, *left_queues);
        }
        retval = 0;
    }
    if (rr < pdata->rd_data->max_routes) {
        if (si_shadow[4000+rr].io.ioseparated.input != 1) {
            if (comp_debug) {
                printf("ERROR: SI did not set reset bit for
                sensor %d\n", 4000+rr);
            }
            retval = 0;
        }
    }
    for (i = 0; i < pdata->rd_data->max_queues; i++) {
        if (qpl[i] != si_qpl[i]) {
            if (comp_debug) {
                printf("ERROR: SI has qpl[%d] at %d while
                  checker has it at %d\n",
                  i, si_qpl[i], qpl[i]);
            }
            retval = 0;
        }
        if ((queues[i] == -1) && (si_queues[i].current_len != 0)) {
            if (comp_debug) {
                printf("ERROR: SI has queue %d not empty while
                  checker has it empty\n", i);
            }
            retval = 0;
        } else if ((queues[i] > -1) && (si_queues[i].routes[0]
          != queues[i])) {
            if (comp_debug) {
                printf("ERROR: SI has queue %d at %d while
                  checker has it at %d\n",
                  i, si_queues[i].routes[0], queues[i]);
            }
            retval = 0;
        }
    }
    return retval;
```

```
}
```

7.7.5.2.5 Zusammenfassung

Die so implementierten Teile der RD-Testprozedur mussten dann noch zusammengefügt werden. Der grobe Ablauf der Testprozedur ist dabei so, dass zunächst die Projektierungsdaten mit Hilfe der in 7.7.5.1 auf Seite 268 hergestellten Funktionen eingelesen werden, um dann daraus, wie oben gezeigt, die konkreten Testfälle zu ermitteln. Danach wird in einer Endlosschleife jeweils eine Testbelegung hergestellt, an SUT und Orakel übergeben, dann die von diesen gelieferten Outputs vergleichen, wobei Gleichheit ein bestandener, Ungleichheit ein nicht bestandener Test ist. Anschließend wird die nächste Testbelegung erzeugt usw. bis alle Belegungen durchlaufen sind.
Darüberhinaus wurden noch einige für den Bediener komfortable, jedoch nicht essentielle Dinge eingebaut. So kann der Bediener zwei Flags setzen, die die Menge der Text-Ausgabe während des Testlaufs von „quasi nichts“ bis „völlige Überflutung“ einstellbar machen. Darüber hinaus kann auch eine einzelne Testbelegung (durch Setzen spezifischer `seed`s) eingestellt und einzeln getestet werden, um z.B. einen bestimmten gescheiterten Test noch einmal mit mehr Ausgaben laufen lassen zu können.
Eine nähere Beschreibung für den Bediener findet sich im Kapitel 7.7.6 auf Seite 298.

Arne Stahlbock

7.7.5.3 Route Controller

An dieser Stelle wird die Implementierung der Testprozeduren für den Route Controller beschrieben. Dieser Vorgang erfolgte in mehreren Etappen:

- Bestimmung einer sinnvollen Menge von Testfällen anhand der RC-Spezifikation
- Implementierung der Funktionalität, aus den Projektierungsdaten diese Testfälle für das spezifische Gleisnetz zu erzeugen und sie in das SUT einzuspeisen (Testgenerator)
- Implementierung der Funktionalität, aus einem Testdatensatz den vom SUT erwarteten Output zu bestimmen (Testorakel)
- Implementierung der Funktionalität, den vom SUT gelieferten Output mit dem des Testorakels zu vergleichen (Testchecker)

7.7.5.3.1 Bestimmung einer Menge von Testfällen

Hierzu betrachten wir zunächst die Parameter der zu testenden Funktion:

```
int process_route(int route, int phase, shm_state_t *shadow,
    struct rc_data_t *rc_data, int *rst, int *bahn_counter,
    int *entry_sensor_counter)
```

Es werden also übergeben:

- die Routennummer
- der Zustand, in dem sich der betreffende RC vor Aufruf befindet
- der aktuelle SHM-Zustand
- die zu diesem RC gehörigen Projektierungsdaten
- die Route State Table
- der Bahnzähler für diese Route
- der Eingangssensorzähler für diese Route

Für die Routennummer bleibt nichts anderes übrig, als alle im Gleisnetz möglichen Routen zu verwenden.
Gleiches gilt für den RC-Zustand: alle Phasen müssen getestet werden.
Beim SHM-Zustand können dagegen wieder Einschränkungen vorgenommen werden. Da der RC für seine Entscheidung nur die Ist-Zustände von Weichen, Sensoren und Signalen sowie Timern beobachtet und dabei auch nur auf das Gegebensein bestimmter Zustände prüft, reicht es aus, für jedes Element lediglich verschiedene gültige Ist-Zustände als Testeingabe zu verwenden. Gegen die Sicherheitsbedingungen verstoßende Zustände werden vom RC nicht beachtet und müssen daher nicht generiert werden. Am einfachsten für die Implementierung ist es, bspw. bei Weichen ungeachtet ihres Typs einfach zwischen den Zuständen links, geradeaus und rechts zu variieren, bei Signalen analog dazu noch die RR-Anzeigen mit einzubeziehen, bei Sensoren die Zähler zu variieren (da hier prinzipiell der Wertebereich nach oben offen steht, wurde eine Begrenzung von 0 bis 4 für die Tests beschlossen) und bei Timern zwischen laufend und nicht laufend zu unterscheiden. Eine weitere Einschränkung ergibt sich dadurch, dass jeweils nur die für die aktuelle Route relevanten Elemente betrachtet werden müssen.
Die Route State Table muss beim RC-Test anders als beim RD nicht komplett durchkombiniert werden, es ist lediglich der für die aktuelle Route geltende Wert zu verändern. Bei den beiden Zählern stünde der Bereich der möglichen Testwerte nach oben offen, daher muss hier eine künstliche Begrenzung vorgenommen werden. Es wurde festgelegt, als Bereich für beide Zähler jeweils die Werte von 0 bis 4 (einschließlich) zu verwenden.

7.7.5.3.2 Testgenerator

Da bei der Implementierung des Testgenerators im Prinzip genauso wie beim RD vorgegangen wurde, soll dem Leser hier eine allzu große Darstellung erspart bleiben. Es wird hier lediglich der Code-Ausschnitt gezeigt, der die für die einzelnen RC-Instanzen relevanten Hardware-Elemente aus den Projektierungsdaten gewinnt und in einer hierfür angelegten Struktur speichert:

```
rcinfo = (struct rc_info*) malloc (sizeof(struct rc_info) *
  pdata->num_rc_data);

// evaluate config data, compute number of test cases
// set gen_xyz values to first test case
for (i = 0; i < pdata->num_rc_data; i++) {
    single_total = 25;
    printf("RC %d\n", i);
    rcinfo[i].num_sensors = 2;
    rcinfo[i].sensor = malloc(sizeof(int) * 2);
    rcinfo[i].sensor[0] = pdata->rc_data[i].entry_sens_id;
    rcinfo[i].sensor[1] = pdata->rc_data[i].exit_sens_id;
    rcinfo[i].gen_sensor = malloc(sizeof(int) * 2);
    rcinfo[i].gen_sensor[0] = 0;
    rcinfo[i].gen_sensor[1] = 0;
    printf("Sensors: %d %d \n", rcinfo[i].sensor[0],
      rcinfo[i].sensor[1]);

    rcinfo[i].num_points = pdata->rc_data[i].num_point;
    single_total *= pow(3, rcinfo[i].num_points);
    rcinfo[i].point = malloc(sizeof(int) * rcinfo[i].num_points);
    rcinfo[i].gen_point = malloc(sizeof(int) *
      rcinfo[i].num_points);
    printf("Points: ");
    for (j = 0; j < rcinfo[i].num_points; j++) {
        rcinfo[i].point[j] = pdata->rc_data[i].point_pos[j].id;
        rcinfo[i].gen_point[j] = 0;
        printf("%d ", rcinfo[i].point[j]);
    }
    printf("\n");

    rcinfo[i].num_signals = pdata->rc_data[i].num_free_sig;
    single_total *= pow(56, rcinfo[i].num_signals);
    rcinfo[i].signal = malloc(sizeof(int) * rcinfo[i].num_signals);
    rcinfo[i].gen_signal = malloc(sizeof(int) *
      rcinfo[i].num_signals);
    printf("Signals: ");
    for (j = 0; j < rcinfo[i].num_signals; j++) {
        rcinfo[i].signal[j] = pdata->rc_data[i].free_sig_pos[j].id;
        rcinfo[i].gen_signal[j] = 0;
        printf("%d ", rcinfo[i].signal[j]);
```

```
        }
        printf("\n");
        printf("test cases: %d\n", (int)(single_total * 1200));
        total += (single_total * 1200);
    }
```

Die Iteration über die verschiedenen Testfälle funktioniert wie beim RC-Test, also etwa:

```
    // increase seed, go to next test case
    gen_rst++;
    if (gen_rst == max_rst) {
        gen_rst = 0;
        j = 0;
        rcinfo[rc].gen_sensor[j] += 1;
        while ((rcinfo[rc].gen_sensor[j] == max_sensor)
          && (j+1 < rcinfo[rc].num_sensors)) {
            rcinfo[rc].gen_sensor[j] = 0;
            rcinfo[rc].gen_sensor[j+1] += 1;
            j++;
        }
        if ((j+1 == rcinfo[rc].num_sensors) &&
          (rcinfo[rc].gen_sensor[j] == max_sensor)) {
            rcinfo[rc].gen_sensor[j] = 0;
            j = 0;
            rcinfo[rc].gen_signal[j] += 1;
            while ((rcinfo[rc].gen_signal[j] == max_signal) &&
              (j+1 < rcinfo[rc].num_signals)) {
                rcinfo[rc].gen_signal[j] = 0;
                rcinfo[rc].gen_signal[j+1] += 1;
                j++;
            }

       [...]
```

7.7.5.3.3 Testorakel

Das Testorakel wurde anhand der RC-Spezifikation in Kapitel 7.5.4.5 auf Seite 191 implementiert. Genau wie beim RD-Testorakel wurde dabei der Originalcode nicht weiter betrachtet, um die Erstellung von identischem Code nicht zu provozieren. Auf eine Code-Wiedergabe kann unserer Meinung nach an dieser Stelle verzichtet werden.

7.7.5.3.4 Testchecker

Auch der Code des Testcheckers soll an dieser Stelle nicht wiedergegeben werden, da er von der Vorgehensweise mit dem gezeigten RC-Testchecker identisch ist und keine spektakulären Innovationen aufweisen kann.

7.7.5.3.5 Zusammenfassung

Die so implementierten Teile der RC-Testprozedur mussten dann noch zusammengefügt werden. Auch hier wurde der vom RD bewährte Ansatz erneut verfolgt. Es wurden auch erneut die für den Bediener hilfreichen Flags und die Möglichkeit, einzelne Testfälle gezielt zu setzen, eingebaut.
Eine nähere Beschreibung für den Bediener findet sich im Kapitel 7.7.6 auf Seite 298.

Arne Stahlbock

7.7.5.4 Safety Monitor

An dieser Stelle wird die Implementierung der Testprozeduren für den Safety Monitor beschrieben. Dieser Vorgang sollte in mehreren Etappen erfolgen, von denen aber nur die erste vollständig, die zweite teilweise erledigt wurde:

- Bestimmung einer sinnvollen Menge von Testfällen anhand der SM-Spezifikation
- Implementierung der Funktionalität, aus den Projektierungsdaten diese Testfälle für das spezifische Gleisnetz zu erzeugen und sie in das SUT einzuspeisen (Testgenerator)
- Implementierung der Funktionalität, aus einem Testdatensatz den vom SUT erwarteten Output zu bestimmen (Testorakel)
- Implementierung der Funktionalität, den vom SUT gelieferten Output mit dem des Testorakels zu vergleichen (Testchecker)

7.7.5.4.1 Bestimmung einer Menge von Testfällen

Hierzu betrachten wir zunächst die Parameter der zu testenden Funktion:

```
int check_safety(shm_state_t *shadow, shm_state_t **realshm,
  struct sm_data_t *sm_data)
```

Es werden also übergeben:

- eine von den RCs ggf. veränderte Kopie des Shared Memory
- der aktuelle Ist-Zustand des Shared Memory

- die zum SM gehörigen Projektierungsdaten (im Wesentlichen die Sicherheitsbedingungen)

Es war von vorn herein klar, dass die Herstellung aller möglichen Eingabekombinationen nicht machbar war, da jedes einzelne SHM-Element 32 Bit umfasst und es davon wiederum in einem größeren Gleisnetz recht viele geben kann.
Es war daher eine sinnvolle Untermenge zu finden. Hierfür wurde das spezifizierte Verhalten der zu testenden Funktion betrachtet: Sie soll einerseits den aktuellen Ist-Zustand auf Sicherheit prüfen, andererseits auch den von den RCs geforderten Soll-Zustand. Nur wenn beide sicher sind und gleichzeitig auch kein Timer abgelaufen ist, wird der Soll-Zustand in das SHM geschrieben, andernfalls erfolgt der Übergang in den Fehlerzustand.
Die Idee war nun folgende:
Es wird über die in den Projektierungsdaten enthaltenen sicheren Zustände iteriert. Für jeden Zustand werden mehrere Testfälle generiert. Einen, in dem dieser Zustand gilt (und der SM ihn folglich akzeptieren muss) und für jedes einzelne Hardware-Element, das in diesem Zustand enthalten ist, Testfälle, die diesen Zustand verletzen (und damit unsicher machen, falls nicht zufällig der so veränderte Zustand seinerseits in der Gesamtliste der sicheren Zustände vorkommt). Einmal würde der Zustand durch eine Verletzung im `real`-SHM, einmal im `shadow`-SHM korrumpiert werden.
Dadurch ergibt sich als Testabdeckung, dass für jeden in den Sicherheitsbedingungen enthaltenen sicheren Zustand einmal ein entsprechender Zustand hergestellt würde, und dass ebenso für jede in diesem enthaltene Teil-Bedingung Verletzungen generiert würden. So kommt man auf eine gewisse Zahl von sicheren wie unsicheren Zuständen, wobei die unsicheren nicht völlig aus der Luft gegriffen, sondern wie einzelne Bitkipper aussehen würden. Darüber hinaus sollten in weiteren Testfällen noch abgelaufene Timer vorgegeben werden, so dass auch dieser Bereich geprüft würde.

Marcin Dysarz, Arne Stahlbock

7.7.5.4.2 Testgenerator

Zu Beginn werden die Projektierungsdaten mit den aus 7.7.5.1 auf Seite 268 bekannten Funktionen eingelesen und einige Datenstrukturen initialisiert. Hierzu zählen diejenigen, die als Eingabe des SUT benötigt werden, sowie einige für den Checker benötigte Strukturen.

Darunter sind zwei, die Anzahl und IDs der in den Sicherheitsbedingungen vorkommenden Elemente aufnehmen sollen:

```
struct hardware_elemente{

  int weichen;
  int sensoren;
```

```
  int signale;
};
```

In dieser Struktur wird die Anzahl der einzelnen HW-Elemente gespeichert.

```
struct hw_elemente_da{

  int *weichen;
  int *sensoren;
  int *signale;
};
```

Hier werden die HW-IDs gruppiert nach Typen (Weichen, Sensoren, Signale) gespeichert. (Zwar wird der Inhalt der zweiten Struktur als solcher nicht mehr benötigt, zum Zeitpunkt der Erstellung war man aber noch davon ausgegangen, und da es nicht schädlich erscheint, hat man sie beibehalten, falls sie doch noch einmal verwendet werden können.)

Diese Strukturen werden dann unter Einsatz einer Hilfsfunktion gefüllt, die alle sicheren Zustände durchläuft und die hier vorkommenden HW-IDs aufnimmt, sofern sie zum ersten Mal auftreten.

Anschließend beginnt der eigentliche Generationsprozess. Dafür werden zunächst folgende Datenstrukturen angelegt:

```
united_conditions = (struct test_shm_value_t *)
    malloc(sizeof(struct test_shm_value_t)
   *(hw_elemente->weichen
     + hw_elemente->sensoren
     + hw_elemente->signale));

united_cond_counter = (struct shm_state_counter *)
    malloc(sizeof(struct shm_state_counter )*1000);

hw_to_test = (struct hw_to_test *)
    malloc (sizeof(struct hw_to_test)
    *(hw_elemente->weichen
      + hw_elemente->sensoren
      + hw_elemente->signale));
```

`united_conditions` ist eine Struktur, die für jedes HW-Element die zugehörige SHM-Belegung aufnehmen soll, `united_cond_counter` nimmt Zählerstände für Sensoren auf und `hw_to_test` soll eine Kopie von `united_conditions` werden, die noch um die zusätzliche Information ergänzt ist, wieviele Bits jeweils im Input- und Output-Bereich gesetzt sind (`xxx_size`):

```
struct hw_to_test{

  int id;
  int input_size;
  int output_size;
  int input;
  int output;
  int input_mask;
  int output_mask;
};
```

Es werden dann `united_conditions` und `united_cond_counter` mit Werten, die dem ersten sicheren Zustand aus den Projektierungsdaten entnommen werden, belegt. Da es prinzipiell vorkommen kann, dass innerhalb eines solchen Zustandes mehrere Bedinungen an ein und dasselbe Element gestellt werden (z.B. kann Bedingung a ein bestimmtes gesetztes Bit bei Element X fordern, Bedingung b ein anderes ebenfalls bei X), werden diese Bedingungen vereinigt, was mit simpler Oder-Verknüpfung der Bitfelder erledigt werden kann. Somit hat man am Ende nur eine Bedingung über jedes Element bezüglich der SHM-Belegung. Bei den Sensorzählern ergibt sich eine solche Vereinigung nicht, da zu jedem Sensor innerhalb eines sicheren Zustandes nur eine Zählerstandsbedingung vorkommt (die Struktur der Projektierungsdaten erlaubt zwar mehr, in der Praxis gibt es aber nur wenige vorkommende Fälle: Sensoren, die nicht Eingangs- oder Ausgangssensor einer aktiven Route sind, müssen auf Null stehen, und bei aktiven Routen muss der Zähler des Eingangssensors größer oder gleich dem des Ausgangssensors sein – zu jedem Sensor also maximal eine Bedingung, da ein Sensor auch niemals zu mehreren gleichzeitig aktiven Routen gehören kann).

Nach Belegung dieser beiden Strukturen wird dann die Kopie in `hw_to_test` erstellt, wozu folgende Hilfsfunktion eingesetzt wird:

```
void copy_mem (){

    for(hardware_loop = 0;
        hardware_loop < united_cond_size;
        hardware_loop++){

          hw_to_test[hardware_loop].id =
            united_conditions[hardware_loop].id;
          ...
         }
    for(bits_loop = 0; bits_loop < 16; bits_loop++){
        ...
```

```
    }
}
```

Die Funktion hat zwei Hauptschleifen. In der ersten werden die in der Struktur `united_conditions` enthaltenen Werte in die Struktur `hw_to_test` kopiert. Die zweite Schleife nimmt eine Teil-Analyse der `united_conditions`-Struktur vor und schreibt in die Struktur `hw_to_test` die Anzahl der einem ID zugehörigen, gesetzten Bits. Anschließend werden mit Hilfe zweier weiterer Funktionen die Daten zur Übergabe an das SUT vorbereitet:

```
void set_shm_shadow()

    for(count_shadow = 0;
        count_shadow < united_cond_size;
        count_shadow++){

        int tmp_id = hw_to_test[count_shadow].id;

          test_shadow[tmp_id].io.ioseparated.input =
    hw_to_test[count_shadow].input;
        ...
    }
for(loop_counter = 0; loop_counter < 1000; loop_counter++){

    test_shadow[(loop_counter+4000)].sensor_counter =
      united_cond_counter[loop_counter].counter;
}
```

Diese Funktion besteht aus zwei Schleifen. In der ersten werden die Werte, die in `struct hw_to_test` enthalten sind, auf `test_shadow` abgebildet. Die zweite Schleife kopiert die Sensorzähler auf das entsprechende `test_shadow`-Bereich.

```
void set_shm_real(){

    for(count_real = 0; count_real < united_cond_size; count_real++){

        int tmp_id = hw_to_test[count_real].id;

        test_real[tmp_id]->io.ioseparated.input  =
            hw_to_test[count_real].input;
        ...
    }
```

```
    for(loop_counter = 0; loop_counter < 1000; loop_counter++){

        test_shadow[(loop_counter+4000)].sensor_counter =
          united_cond_counter[loop_counter].counter;
    }
}
```

Diese Funktion hat identischen Aufbau und Funktion wie die obere, mit einem Unterschied, nämlich dass sie die in `struct hw_to_test` gespeicherten Werte auf `test_real` abbildet.

Die Datenstrukturen, die an den Testling zu übergeben sind, sind nun belegt, so dass der Testling aufgerufen werden kann. An dieser Stelle müsste dann auch der Einsatz von Orakel und Checker erfolgen.

Nach diesem ersten Testfall, der mit den Belegungen durchgeführt wurde, die dem sicheren Zustand entsprechen, müssen nun solche generiert werden, die diesen Zustand verletzen. Dabei wird zunächst das bereits beschriebene `copy_mem()` ausgeführt, um die Kopie in `hw_to_test` wieder dem Original anzugleichen (nach diesem ersten Test sollte da zwar nichts verändert sein, nach den weiteren wird das aber der Fall sein).

Ab dieser Stelle hat der Testvorgang dasselbe Schema: Speicher kopieren, einen Wert verfälschen, auf `test_shadow` und `test_real` abbilden und den Testling aufrufen. Es wird nun der Verfälschungs-Algorithmus beschrieben:

```
int test_interrationen = united_cond_size;

for(; test_loop < test_interrationen; test_loop++){

}
```

Erstmal wird festgestellt, wie viele Testiterationen für einen sicheren Zustand gemacht werden müssen. Diese Zahl ist gleich der Anzahl der HW-Elemente. Die Schleife hat mehrere Zweige, die ähnlich arbeiten. Diese werden hier nicht komplett dargestellt. Ein Zweig wird als Beispiel beschrieben.

```
int tmp_id_to_test = hw_to_test[test_loop].id;
if((tmp_id_to_test >= 1000) &&(tmp_id_to_test < 2000)){
}
```

Zunächst wird die erste HW-ID aus der Struktur `hw_to_test` geholt und es wird geprüft, was für ein Hardware-Element vorliegt (eine Weiche, ein Signal, ein Sensor).

```
if(hw_to_test[test_loop].input_size == 0){

    if(hw_to_test[test_loop].input_mask == 0){
    \\ tue nichts
    }
    else {
       for(i = 0; i < 16; i++){

           int input_mask_tmp = hw_to_test[test_loop].input_mask;

           if((input_mask_tmp)&(1<<i)){

              int setze_bit = i;
              hw_to_test[test_loop].input = pow(2,setze_bit);
           }
       }
    }
else {
\\ ein input Bit gesetzt, dann loeche

for(i = 0; i < 16; i++){

    int input_mask_tmp = hw_to_test[test_loop].input_mask;

    if((input_mask_tmp)&(1<<i)){

       int setze_bit = i;
       printf("\n setze_bit: %d   %d\n ",setze_bit, pow(2,setze_bit));
       hw_to_test[test_loop].input -= pow(2,setze_bit);
       break;
    }
    if(hw_to_test[test_loop].output_size == 0){

       if(hw_to_test[test_loop].output_mask == 0){
          \\ tue nichts
       }
       else {
         \\ analog wie beim input
       }
     }
     else {
```

```
        \\ analog wie beim input
      }
    }
}
```

Mit diesen Abfragen wird festgestellt, wie viele Bits im Input- bzw. Output-Bereich gesetzt sind und was für Bit-Masken gültig sind. Wenn ein Bit in Input- oder Output-Bereich gesetzt ist, dann wird es gelöscht und die Schleife wird abgebrochen. Wenn die Bit-Masken so gesetzt sind, dass kein Bit überhaupt betrachtet wird, dann wird nichts gemacht. Wenn die Bit-Masken nicht Null sind und kein Bit in Input- oder Output-Bereich gesetzt ist, dann wird das erste von der Bit-Maske betrachtete Bit gesetzt und die Schleife wird abgebrochen.

Der so verletzte Zustand wird dann, nachdem `hw_to_test` auf `test_real` und `test_shadow` abgebildet wurde, dem Testling als nächster Testfall übergeben.

Danach wird `hw_to_test` wieder mittels `copy_mem()` dem Original des sicheren Zustandes angeglichen und der Verfälschungsalgorithmus auf das nächsten HW-Element in diesem Zustand angewandt.

Ist letztlich für jedes HW-Element eine Korrumpierung des sicheren Zustandes durchgeführt worden, beginnt der übergeordnete Ablauf mit dem nächsten sicheren Zustand aus den Projektierungsdaten. Es werden dann neue `united_conditions` gebildet usw.

Was in diesem Algorithmus noch fehlt, ist die Verfälschung von Zählerwerten, so dass auch auf diesem Wege unsichere Zustände hergestellt werden. Ebenso werden noch keine abgelaufenen Timer benutzt, um den SM auf die in diesem Fall geforderte Reaktion zu prüfen.
An dieser Stelle ist der Programmiervorgang beendet worden. Test-Orakel und -Checker existieren nicht.

Marcin Dysarz, Arne Stahlbock

7.7.6 Testausführung und -auswertung

7.7.6.1 Generelles

Es wurde ein Projekt für den RT-Tester erstellt, in das die fertigen Testprozeduren (also diejenigen für RD und RC) aufgenommen wurden. Dieses Projekt ist wie folgt strukturiert:

- Unterverzeichnis `RTC/test` – enthält die Testprozeduren für den RC

- Unterverzeichnis `RTD/test` – enthält die Testprozeduren für den RD
- Unterverzeichnis `SUT/RTC` – enthält den zu testenden RC
- Unterverzeichnis `SUT/RTD` – enthält den zu testenden RD
- Unterverzeichnis `stubsrc` – enthält Hilfsfunktionen für Testprozeduren

Im Wurzelverzeichnis der Testsuite befinden sich außerdem mehrere Konfigurationsdateien. Besonderes Augenmerk ist dabei auf `tracstest.rtg` zu richten, da diese Datei für jedes System bzw. jeden Benutzer, der die Testsuite einsetzen möchte, verändert werden muss. Beispielhaft kann der fragliche Abschnitt dieser Datei wie folgt aussehen:

```
<ui-config>
 <commands>
  <command path="/usr/bin/rtt-compile-test"
    name="/RTT-GUI/Command/Compile" />
  <command path="/usr/bin/rtt-new-test"
    name="/RTT-GUI/Command/New Test Case" />
  <command path="/usr/bin/rtt-copy-test"
    name="/RTT-GUI/Command/Copy" />
  <command path="/usr/bin/rtt-run-test"
    name="/RTT-GUI/Command/Run" />
  <command path="/usr/bin/rtt-stop-test"
    name="/RTT-GUI/Command/Stop" />
  <command path="/usr/bin/rtt-doc-test"
    name="/RTT-GUI/Command/Doc" />
  <command path="" name="/RTT-GUI/Command/Index" />
  <command path="" name="/RTT-GUI/Command/Coverage" />
 </commands>
 <environment>
  <env-var value="/usr" name="RTTHOME" />
  <env-var value="/home/wolf359/tracs/cvs/hwsw/testsuite"
    name="RTT_TESTCONTEXT" />
 </environment>
</ui-config>
```

Hier sind die gegebenen Werte für `command path` den Pfaden auf dem jeweiligen System anzupassen, damit die RT-Tester-GUI die Binaries finden kann. Außerdem sind die beiden Umgebungsvariablen `RTTHOME` und `RTT_TESTCONTEXT` anzupassen. Wurde die Anpassung vorgenommen, kann die RT-Tester-GUI gestartet werden. Zur Bedienung des RT-Testers und der GUI wird an dieser Stelle auf das entsprechende Manual [Ver04] verwiesen.

7.7.6.2 Route Dispatcher

7.7.6.2.1 Ausführung

Um den RD-Test durchführen zu können, muss zunächst die Datei `RTD/test/specs/director.rts` angepasst werden. Relativ am Anfang muss die Zeile

```
const char *filename = "/home/wolf359/tracs/cvs/compiler/code/final/pd";
```

auf die zu verwendende Projektierungsdaten-Datei angepasst werden.
Am Anfang der Hauptprozedur `abstract machine director_main()` können zudem weitere Einstellungen vorgenommen werden. Gesetzt werden können die Werte:

- `debug` – auf 1 gesetzt, erfolgen mehr Ausgaben bei der Generierung der Testfälle
- `comp_debug` – auf 1 gesetzt, erfolgen mehr Ausgaben beim Vergleich der Outputs von SUT und Orakel
- `single_test` – auf 1 gesetzt, erfolgt kein kompletter Durchlauf aller Testfälle, sondern nur ein einzelner Test

In dem direkt folgenden Abschnitt können dann die Werte für diesen einzelnen Test gesetzt werden. Beispielhaft könnte das so aussehen:

```
single_gen_queues = (int *) malloc(sizeof(int) * 4);
single_gen_queues[0] = 0;
single_gen_queues[1] = 0;
single_gen_queues[2] = 0;
single_gen_queues[3] = 2;

single_gen_qpl = 8;
single_gen_rst = 179;
single_gen_left_queues = 1;
single_gen_rr = 0;
```

Die hier gesetzten Werte werden dann als `seed` für die jeweiligen Variablen benutzt (siehe auch die Beschreibung in 7.7.5.2 auf Seite 277). Was die einzelnen Werte letztlich für Eingaben bewirken, kann man bei Setzen von `debug` auf 1 erfahren, da dann bei der Generierung mehr Ausgaben erfolgen. (Anmerkung: Schlägt ein Testfall fehl, werden die für diesen Testfall verantwortlichen `seed`-Werte ausgegeben, damit sie ggf. noch einmal gezielt gesetzt werden können.) Der Pointer `single_gen_queues` muss so viele Elemente erhalten, wie es Queues in dem Test-Gleisnetz gibt.
Es muss jedoch darauf hingewiesen werden, dass die Ausführung einzelner Testfälle und vor allem die Interpretation ihrer Ausgaben nicht trivial ist und daher nur mit genügend Hintergrundwissen um die SI-Spezifikation und die Testprozeduren sinnvoll ist. Für den

Laien muss die Ausführung der kompletten Testreihe (`single_test = 0`) genügen – ist sie erfolgreich, so ist der getestete RD in Ordnung; schlägt sie fehl, so muss er sich mit eben diesem Ergebnis zufriedengeben und zur Ermittlung der Fehlerursache Experten (z.B. TRACS) hinzuziehen.

7.7.6.2.2 Auswertung

Der uns anfangs vorliegende RD wurde mit den Test-Gleisnetzen 1 und 2 getestet, dabei wurden zwei Fehler zu Tage gefördert:

- In zwei aus dem Zustand „Route bearbeiten“ ausgehenden Transitionen verlangte die Spezifikation einen Vergleich von `uebersprungene_queues` mit `anz_queues`. Hier war in der RD-Implementation der Vergleichsoperator falsch (die Spezifikation verlangt >= und <, es waren > und <= implementiert).
- In der dritten aus „Route bearbeiten“ ausgehenden Transition (wenn eine Route freigeschaltet wird) wird laut Spezifikation kein Vergleich von `uebersprungene_queues` mit `anz_queues` vorgenommen. In der Implementation geschah das aber doch.

Diese Fehler wurden der SI-Gruppe gemeldet und von ihr behoben, so dass der RD nunmehr keine bekannten Fehler mehr aufweist.

Arne Stahlbock

7.7.6.3 Route Controller

7.7.6.3.1 Ausführung

Um den RD-Test durchführen zu können, muss zunächst die Datei `RTC/test/specs/director.rts` angepasst werden. Relativ am Anfang muss die Zeile

```
const char *filename = "/home/wolf359/tracs/cvs/compiler/code/final/pd";
```

auf die zu verwendende Projektierungsdaten-Datei angepasst werden.
Am Anfang der Hauptprozedur `abstract machine director_main()` können zudem weitere Einstellungen vorgenommen werden. Gesetzt werden können die Werte:

- `debug` – auf 1 gesetzt, erfolgen mehr Ausgaben bei der Generierung der Testfälle
- `comp_debug` – auf 1 gesetzt, erfolgen mehr Ausgaben beim Vergleich der Outputs von SUT und Orakel
- `single_test` – auf 1 gesetzt, erfolgt kein kompletter Durchlauf aller Testfälle, sondern nur ein einzelner Test

In dem direkt folgenden Abschnitt können dann die Werte für diesen einzelnen Test gesetzt werden. Beispielhaft könnte das so aussehen:

```
single_gen_rst = 1;
single_gen_bc = 0;
single_gen_esc = 0;
single_gen_phase = 0;
single_gen_route = 8;
single_gen_timer = 0;
single_gen_shadow->gen_point[7] = 0;
single_gen_shadow->gen_point[2] = 0;
single_gen_shadow->gen_signal[1] = 0;
single_gen_shadow->gen_sensor[102] = 0;
single_gen_shadow->gen_sensor[104] = 0;
```

Die hier gesetzten Werte werden dann als `seed` für die jeweiligen Variablen benutzt (siehe auch die Beschreibung in 7.7.5.3 auf Seite 287). Was die einzelnen Werte letztlich für Eingaben bewirken, kann man bei Setzen von `debug` auf 1 erfahren, da dann bei der Generierung mehr Ausgaben erfolgen. (Anmerkung: Schlägt ein Testfall fehl, werden die für diesen Testfall verantwortlichen `seed`-Werte ausgegeben, damit sie ggf. noch einmal gezielt gesetzt werden können.) Die Indizes bei `gen_point` sind die Weichen-ID minus 1000, bei Signalen entsprechend minus 3000, Sensoren minus 4000. Kreuzungsweichen werden bei den Weichen eingegeben und haben dann Indizes von 1000 aufwärts.
Es muss jedoch darauf hingewiesen werden, dass die Ausführung einzelner Testfälle und vor allem die Interpretation ihrer Ausgaben nicht trivial ist und daher nur mit genügend Hintergrundwissen um die SI-Spezifikation und die Testprozeduren sinnvoll ist. Für den Laien muss die Ausführung der kompletten Testreihe (`single_test = 0`) genügen – ist sie erfolgreich, so ist der getestete RD in Ordnung; schlägt sie fehl, so muss er sich mit eben diesem Ergebnis zufriedengeben und zur Ermittlung der Fehlerursache Experten (z.B. TRACS) hinzuziehen.

7.7.6.3.2 Auswertung

Der uns anfangs vorliegende RC wurde mit den Test-Gleisnetzen 1 und 2 getestet, dabei wurden folgende Fehler zu Tage gefördert:

- Es war zwar spezifiziert, dass der Eingangssensorzähler zu den Ein- und Ausgabedaten des RC gehört, es war aber nicht spezifiziert, wo und unter welchen Umständen er geändert wird. In der Implementation wurde er in `eingangssensor_ausgeloest` geändert.
- In `eingangssensor_ausgeloest`: Der dort durchgeführte Vergleich sollte besser mit $<$ statt mit $!=$ durchgeführt werden, ansonsten könnte `entry_sensor_count` ins Unermessliche wachsen

- In `eingangssensor_ausgeloest`: Das Erhöhen des Wertes funktioniert nur mit `(*entry_sensor_count)++`, nicht mit `*entry_sensor_count++`.
- In der Deaktivierungsphase wich die Implementation von der Spezifikation ab: Die Transition zur Testphase hat u.a. als Bedingung, dass der RST-Eintrag == `ROUTE_SPERREN` sein muss, diese Bedingung kam im Code nicht vor.

Diese Fehler wurden der SI-Gruppe gemeldet und von ihr behoben, so dass der RC nunmehr keine bekannten Fehler mehr aufweist.

Arne Stahlbock

7.7.6.4 Safety Monitor

Wir können nicht sagen, dass wir eine komplette, automatische Testausführung und Auswertung gemacht haben. Dazu fehlte uns der Test-Checker. Aber während der Programmierungs-Phase haben wir die Testroutine mehrmals durchgeführt und „manuell“ (auf Grund der Ausgabe) ausgewertet. Da haben wir bemerkt, dass bei dem ersten Testdurchlauf, wenn noch die Input-/Output-Werte nicht korrumpiert sind, die zu testende Funktion antworte trotzdem negativ (hat also die Input-/Output-Belegung als ungultig).
An dieser Stelle sollte man genau untersuchen, woran es liegt und den Test-Checker programmieren. Wenn er genauso wie zu testende Funktion arbeiten soll, würde es einfacher eine Entscheidung zu treffen, woran die Fehler liegen.

Marcin Dysarz

7.7.7 Vorschläge zu Software–, Hardware/Software– und System Integrationstest

In diesem Abschnitt werden die Konzepte für Software Integrationstest, Hardware/Software Integrationstest und System Integrationstest entwickelt. Innerhalb des Projekts wurden ausschliesslich Unit Tests durchgeführt. Um aber korrektes Funktionieren in Bezug auf die Spezifikation vollständig zu testen, sind die genannten weiterführenden Tests notwendig, um das Zusammenspiel von einzelnen Komponenten und Zielhardware miteinander zu überprüfen.
Alle drei Testarten beruhen auf der in 7.5 vorgestellten Spezifikation des Safety Controllers. Für die dabei benutzten *UML Statecharts* existieren Testtheorien, die das Testen gegen eine solche Spezifikation ermöglichen. Hierzu sind einige Vorgaben an den Testling notwendig, die später beschrieben werden sollen. Zusätzlich muss diese Spezifikation noch transformiert werden, um Tests zu ermöglichen. Wird auf solche Weise getestet, so ermöglicht das benutzte Testwerkzeug, dass ein Testgenerator für sowohl Software

Integrationstest, als auch Hardware/Software Integrationstest notwendige Testdaten generiert. Hierbei werden die selben Testdaten auf unterschiedliche Weise an den Testling weitergereicht, ohne dass dies Einfluss auf die Testfallgenerierung hat. Ebenso wird für Software Integrationstest und Hardware/Software Integrationstest nur ein Testchecker benötigt, da auch dieser seine Informationen in abstrakter Form erhält.
Zunächst soll nun auf die benutzte Testtheorie sowie die dazu notwendigen Testkomponenten eingegangen werden, bevor deren Anwendung innerhalb der einzelnen Testarten beschrieben wird.

Helge Löding

7.7.7.1 Spezifikation

Die in 7.5 vorgestellte Spezifikation des TRACS Steuerinterpreters bildet in sich noch keine Spezifikation für einen konkreten Safety Controller, da hier insbesondere nicht klar ist, welche Anzahl von Routen zu steuern sind. Da pro Route ein Route Controller notwendig ist, kann dort nur das Verhalten eines einzelnen Route Controllers abstrakt beschrieben werden. Zusammen mit der Projektierung eines solchen Steuerinterpreters ergibt sich erst ein konkreter Safety Controller. Daraus folgt, dass diese Spezifikation zu Testzwecken noch um Projektierungsinformationen ergänzt werden muss.
Um einen Safety Controller gegen seine Spezifikation testen zu können, muss das Testsystem diese Spezifikation also aus der Spezifikation des Steuerinterpreters und den zugehörigen Projektierungsdaten für einen Safety Controller herleiten. Hierbei muss aus den Projektierungsdaten ermittelt werden, wieviele Routen zu steuern sind, und die abstrakte Spezifikation eines Route Controllers für jede Route in konkrete Spezifikationen für jeden zu benutzenden Route Controller des Safety Controllers umgesetzt werden. Die Spezifikationen von Route Dispatcher und Safety Monitor des Steuerinterpreters können auch als Spezifikationen derselben Komponenten innerhalb des Safety Controllers angesehen werden, da sie immer nur einfach existieren und in beiden Spezifikationen direkt auf den Projektierungsdaten arbeiten.
Um gegen eine so entstandene Spezifikation eines Safety Controllers testen zu können, muss sie allerdings noch weiter transformiert werden. Die entstandene Spezifikation ist ebenso wie die Spezifikation des TRACS Steuerinterpreters formal hierarchisch aufgebaut. Dies bereitet Probleme, da die zu verwendende Testtheorie flache (nicht hierarchische nicht parallele) Statecharts benötigt. Es müssen also alle Hierarchieebenen der Spezifikation auf eine einzelne Hierarchieebene reduziert werden. Dies ist im Allgemeinen nicht trivial, kann allerdings für unsere Spezifikationen gut automatisiert werden.
Auf oberster Ebene eines Safety Controllers befinden sich Zustände für Route Dispatcher, alle existierenden Route Controller und Safety Monitor. Zwischen diesen Zuständen wird zyklisch gewechselt, wobei dies von einer Zustandsvariable gesteuert wird, die innerhalb der einzelnen hierarchischen Zustände der einzelnen Komponenten verändert wird. Um diese Hierarchie aufzulösen, werden jetzt alle Zustände der Spezifikation in

ein einzelnes Statechart übernommen. Jetzt müssen von jedem Zustand des Route Dispatchers Übergänge zum ersten Route Controller eingefügt werden, die der Transition auf der ursprünglich obersten Ebene entsprechen. Analog müssen Übergänge von jedem Route Controller n zum Route Controller $n + 1$ eingefügt werden. Ebenso müssen Übergänge vom letzten Route Controller zum Safety Monitor eingefügt werden.
Da innerhalb der Hierarchie der ursprünglichen Spezifikation auch History (Einnahme eines zuletzt aktuellen Zustandes innerhalb einer Hierarchiestufe) benutzt wird, muss auch hier eine Transformation vorgenommen werden. Es muss hierzu bei jedem Zustandswechsel zwischen ursprünglich hierarchischen Komponenten in der flachen Entsprechung Übergänge zu jedem Zustand der Zielkompontente geben, wobei diese Übergänge durch ein Zustandsfeld pro Komponente unterschieden werden müssen. Dieses Zustandsfeld muss bei jedem Übergang innerhalb einer Komponente aktualisiert werden.
Zusammengefasst ergibt dies folgendes Vorgehen:

- Zustände aller Safety Controller Komponenten in ein einzelnes Statechart übernehmen
- Bei jedem Übergang innerhalb einer Komponente ein entsprechendes Zustandsfeld verwalten
- Von jedem Zustand einer Komponente Übergänge zu jedem Zustand der Folgekomponente anlegen
- Solche Übergänge mit Bedingungen und Aktionen aus ursprünglich oberster Ebene versehen
- Solche Übergänge zusätzlich mit Bedingungen über das Zustandsfeld der Zielkomponente versehen

Helge Löding

7.7.7.2 W-Methode

Die oben beschriebene flache Spezifikation eines konkreten Safety Controllers kann nun als Basis für Testfallgenerierung genutzt werden. Hierbei kommt es darauf an zu zeigen, dass die Implementation des Safety Controllers die selben Zustände und Übergänge wie die Spezifikation hat. Hierzu wird die sogenannte W-Methode aus [Cho78] verwendet, mit der diese Forderungen nachgewiesen werden können, indem nacheinander alle Zustände der Spezifikation angenommen werden, und dann Eingaben geliefert werden, die durch Reaktionen des Testlings darauf den aktuellen Zustand von allen anderen Zuständen unterscheiden. Hierzu sind allerdings Erweiterungen notwendig, da bei der original W-Methode nur Testtraces aus Events generiert werden. Da wir in unserer Spezifikation allerdings nur mit Bedingungen und nicht mit Events arbeiten. Dies lässt sich allerdings aufeinander abbilden, wie später beschrieben wird.

Zusätzlich können nicht alle Übergänge der Spezifikation erzwungen werden, da nicht-intrusiv getestet werden soll, und nicht alle Übergänge von externen Bedingungen abhängen. Hierzu müssen Testtraces zusätzlich mit dem internen Alphabet (den möglichen intern auftretenden Ereignissen) kombiniert werden, um jeden Zustand der Spezifikation erreichen zu können. Ein genaues Vorgehen muss hier noch entwickelt werden.

Helge Löding

7.7.7.3 System under Test

Um die W-Methode aus [Cho78] anwenden zu können ist es nötig, dass der Testling in seinen Startzustand versetzt werden kann, da jeder Testfall vom Startzustand ausgeht. Hierzu sind entsprechende Mechanismen noch zu entwickeln.
Zunächst wird vorgeschlagen, dass der Testling vom Testsystem im laufenden Betrieb abgebrochen und für jeden Testfall neu gestartet wird. Dies kann aber negative Folgen für die Durchführbarkeit aller Tests haben, da sich deren Laufzeit hierdurch deutlich verlängern kann.
Alternativ wäre ein Reset-Mechanismus innerhalb der Testlings vorstellbar. Dies wäre allerdings eine Massnahme, die einen Eingriff in den Testling erfordern und somit intrusives Testen bedeuten würde.

Helge Löding

7.7.7.4 Testgenerator

Um Integrationstests durchführen zu können, wird ein Testgenerator entwickelt, der als Eingabe die wie oben beschrieben hergeleitete flache Spezifikation eines Safety Controllers erhält. Aus dieser Spezifikation gehen dann mithilfe der W-Methode ([Cho78]) Eingaben an den Testling hervor, die vollständige Zustands- und Übergangsüberdeckung der Spezifikation des Testlings garantiert. Da der Testling aus Sicht von Integrationstests als Eingaben nur den Ist-Zustand eines Gleisnetzes in Form von Shared-Memory Daten erhält, werden Stimulationen des Testlings vom Testgenerator entsprechend in den Ist-Zustand Bereich des Shared-Memory geschrieben. Art und Reihenfolge der Stimulationen bestimmen hierbei den Zustand, in den der Testling übergeht.
Da die Spezifikation nicht auf Events basiert, sondern durch Bedingungen über Shared-Memory Werte gesteuert wird, sind zusätzliche Funktionen notwendig, die in Abhängigkeit der Projektierung eines Safety Controllers Alle Shared-Memory Bedingungen der Spezifikation wahr oder falsch werden lassen bzw. das Shared-Memory entsprechend modifizieren. Diese Funktionen entsprechen dann den in [Cho78] verwendeten Events einer Spezifikation. Ein Testtrace ist dann eine Folge von Aufrufen dieser Funktionen.
Um auch interne Übergänge des Testlings überdecken zu können, muss das interne Alphabet des Testlings bekannt sein. Dies sind für einen TRACS Safety Controller insbesondere die internen Zustandsfelder für Befahrungskonstellationen der einzelnen Routen.

Diese können nur intrusiv stimuliert werden, und sollten daher aus der Spezifikation in jedem Tesfall impliziert werden.

Helge Löding

7.7.7.5 Testchecker

Um die Reaktionen eines Testlings auf die Stimulationen des Testgenerators prüfen zu können, ist ein Testchecker notwendig, der die Ausgaben des Testlings überwacht, und sie auf Inkonsistenzen zur zugrundeliegenden Spezifikation überprüft. Hierzu wird die wie oben beschrieben hergeleitete Spezifikation innerhalb des Testcheckers simuliert. Der Checker verarbeitet die Eingaben des Testgenerators ebenso wie der Testling selbst, und verändert innerhalb der Simulation seinen Zustand gemäss der Spezifikation. Sind laut Spezifikation Aktionen des Testlings zu erwarten, so prüft der Testchecker den Soll-Zustand des Shared-Memory vom Testling und prüft damit, ob sich der Testling spezifikationsgemäss verhält. Hierzu sind also für alle Shared-Memory Aktionen der Spezifikation zusätzliche Funktionen notwendig, die jede Shared-Memory Aktion des Testlings in Abhängigkeit seiner Projektierung überprüfen. Diese Funktionen ersetzen in der Simulation der Spezifikation die eigentlichen Aktionen, da der Testchecker nur passiv wirken soll.
Der Testchecker muss allerdings nicht nur die Eingaben des Testgenerators verarbeiten, sondern auch die internen Zustandsfelder wie Befahrungskonstellationen einzelner Routen verwalten. Da diese aus Sicht von Integrationstests nicht sichtbar sind, können sie innerhalb des Testcheckers nicht im Testling eingesehen werden und müssen dementsprechend als eigene Kopie zu Vergleichszwecken selbst verwaltet werden.

Helge Löding

7.7.7.6 Integrationstests

Im Software Integrationstest werden nun alle Softwarekomponenten auf korrektes Zusammenspiel geprüft. Hierbei laufen Testumgebung und Testling auf dem selben Rechner, da hierbei nur Softwareaspekte der Integration interessant sind.
Mit dem oben beschriebenen Testgenerator und Testchecker kann nun ein Software Integrationstest durchgeführt werden. Hierbei werden Projektierungsdaten für einen konkreten Safety Controller eingelesen und dessen flache Spezifikation hergeleitet. Der Testgenerator kann daraus nun Testtraces generieren und ausführen. Dadurch wird der Testling durch Schreiben von Shared-Memory Werten im Ist-Zustand stimuliert und muss reagieren. Der Testchecker simuliert die Spezifikation und überprüft die resultierenden Shared-Memory Werte im Soll-Zustand. Treten Inkonsistenzen zur Spezifikation zwischen Testling und Simulation auf, so gilt der Test als gescheitert. Abbildung 7.59 auf der nächsten Seite zeigt das Zusammenspiel von Spezifikation, Testgenerator, Testchecker und Testling im Testaufbau.

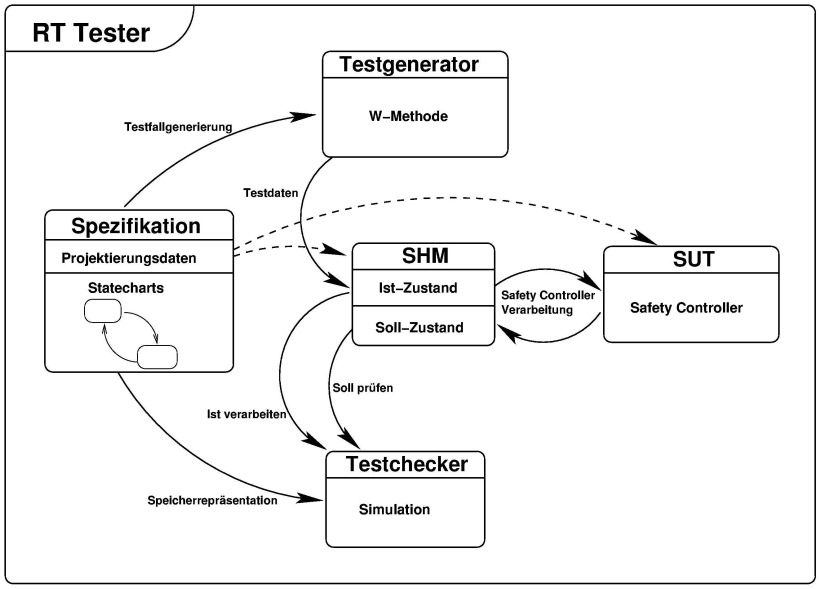


Abbildung 7.59: Testkonzept der Software Integrationstests

Nun kann ein Hardware/Software Integrationstest durchgeführt werden. Hierbei wird die zu testende Software auf ihrer Zielhardware getestet. Das Testvorgehen entspricht hierbei dem Vorgehen im Software Integrationstest. Aufgrund der Architektur der Testumgebung bleiben hierbei Testgenerator und Testchecker unverändert. Zwar müssen hierbei Ein- und Ausgaben des Testlings über I/O Hardware vom und an das Testsystem umgeleitet werden, dies geschieht allerdings innerhalb des *Hardware Abstraction Layer*, der Schicht des Testsystems, die Ein- und Ausgaben des Testlings mit Testkomponenten verbindet. Abbildung 7.60 auf der nächsten Seite zeigt den Aufbau eines Hardware/Software Integrationstests.
Zuletzt wird ein System Integrationstest durchgeführt. Hierbei bleibt der Testgenerator aussenvor, da hier keine Testdaten mehr generiert werden, sondern ein echtes Gleisnetze an den Testling angeschlossen wird. Der Testchecker bleibt unverändert, simuliert weiterhin die Spezifikation der Safety Controllers, und überprüft dadurch dessen spezifikationsgemässes Verhalten. Abbildung 7.61 auf Seite 310 zeigt den Testaufbau.

Helge Löding

7.7.7.7 Erwägungen zu Laufzeit und Timing

Da insbesondere bei vielen Routen die flache Spezifikation eines Safety Controllers relativ viele Zustände enthält ist davon auszugehen, dass nicht alle Tests in angemessener Zeit ausgeführt werden können. Um dieses Problem zu lösen, muss eine geeignete Auswahl von Testfällen getroffen werden. Hierzu ist eine Heuristik notwendig, die die

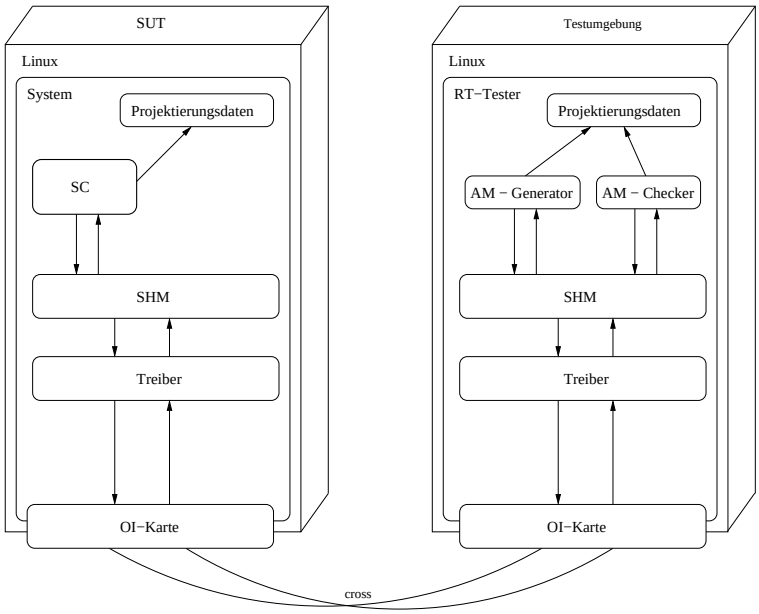


Abbildung 7.60: Schema des Hardware/Software Integrationstests

aus Sicherheitsaspekten relevanten Testfälle aussucht. Hierbei sollten insbesondere die sicherheitskritischen Übergänge der Transition geprüft werden. Eine solche Heuristik wurde nicht entwickelt.
Desweiteren ist in Zukunft eine Erweiterung des oben beschriebenen Testvorgehens notwendig, um auch harte Echtzeiteigenschaften eines Safety Controllers überprüfen zu können. Hierzu könnten die in [AD94] vorgeschlagenen *Grid Automata* benutzt werden. Dies steht allerdings nicht im Widerspruch zum bisherigen Testkonzept, da auch *Grid Automata* eine Erweiterung der in [Cho78] eingeführten W-Methode sind. Ein solches Konzept zum Testen von Echtzeiteigenschaften eines Safety Controllers wurde ebenfalls noch nicht entwickelt.

Helge Löding

7.7.8 Reflexion

In diesem Kapitel sollen die Erkenntnisse und Erfahrungen über die Arbeit, die für die Entwicklung der Testprozeduren geleistet wurde, eingeschätzt und bewertet werden.

7.7.8.1 Wintersemester 2003/2004

In dem ersten Projektsemester wurden theoretische Grundlagen des Testvorganges erfasst. Es wurde ein Arbeitsplan erstellt, der die Arbeitspakete enthielt, die im nächsten Semester in die Tat umgesetzt werden sollten.

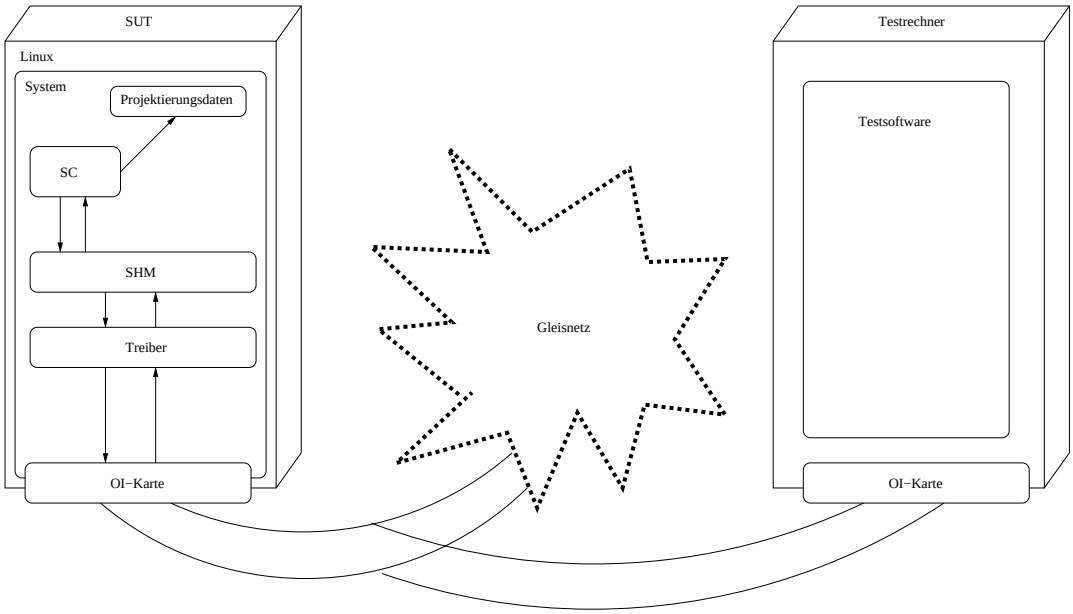


Abbildung 7.61: Schema des System Integrationstests

7.7.8.2 Sommersemester 2004

In diesem Semester sollten die Arbeitspakete, die im vorangegangenen Semester geplant worden sind, erfüllt werden. Leider war das aus verschiedenen Gründen nicht möglich. Eine Ursache dafür war, dass die Entwickler kein Produkt, das testbereit wäre, geliefert haben (auch keine Alpha-Version) und zusätzlich auch nur eine vorläufige, für die Tests nicht ausreichende Spezifikation vorlag. Eine andere Ursache lag bei der falschen Planung und Arbeitsaufteilung.

7.7.8.3 Wintersemester 2004/2005

Nachdem sich einige Projektteilnehmer nach der Benotung ihrer Arbeit in dem vorigen Semester verabschiedet haben, ist die Testgruppe mit einer Besatzung von einem Mann verblieben. Das System war immer noch nicht testbereit. Man versuchte dann die Testgruppe immer irgendwo anders einsetzen, wo momentan etwas zu tun war. Die Gruppe beteiligte sich bei der Bestimmung der Systemgrenzen, hat Auskunft über in der Branche verfügbare Hardware eingeholt, und infolgedessen sind auch Beispielgleisnetze entstanden. Das alles waren aber Arbeiten, die praktisch nichts mit den eigentlichen Tests zu tun hatten. Theoretisch war es auch nicht möglich, eine vollständige Testprozedur zu planen, weil die Systemspezifikation nicht als endgültig erklärt worden ist.

7.7.8.4 Sommersemester 2005

Relativ zu Beginn des Semesters wurde die Gruppe wieder um eine Person erweitert, da durch Fertigstellung eines anderen Teilprojekts Arbeitskraft frei geworden war. Es herrschte jedoch nach wie vor der Zustand, dass die benötigte Spezifikation nicht vorlag. Es musste daher auf Arbeiten ausgewichen werden, die zu diesem Zeitpunkt schon erledigt werden konnten, wie z.B. die Implementierung des Projektierungsdaten-Einlesens (glücklicherweise war dieses Format schon stabil). Auch wurde die Fertigstellung der SI-Spezifikation durch Reviews derselben unterstützt.

Darüber hinaus fiel in diese Phase auch die Vorbereitung des Projekttages, in die einiges (wenn auch nicht alles) an Kapazität floss, was im Nachhinein auch als berechtigt angesehen werden muss. Erst kurz vor dem Projekttag konnte, als die SI-Spezifikation fertiggestellt war, mit der Implementierung der Testprozeduren begonnen werden.

Bei der Aufwandsschätzung für diese hatte man dann noch gewaltig danebengelegen, man muss diese Prognosen im Rückblick als unerfüllbares Wunschdenken betrachten.

Nach dem Projekttag kam zudem auch noch die Zeit, in der der Abschlussbericht vorbereitet werden musste, so dass hier wiederum Zeit anderweitig eingesetzt werden musste, die dann natürlich der Implementierung verlorenging. Es musste so letztlich oft an mehreren Baustellen gleichzeitig gearbeitet werden, wenn es besser gewesen wäre, sich auf eine Sache konzentrieren zu können.

Es konnten somit nur noch die Testprozeduren für den Route Dispatcher und den Route Controller fertiggestellt und die jeweiligen Tests durchgeführt werden, wobei auch in der Tat einige Fehler der getesteten Systeme zum Vorschein kamen.

Es stand letztlich nur knapp über ein halbes Semester für die angesetzten Arbeiten zur Verfügung, wobei davon auch noch ein gewisser Teil in andere Bereiche von TRACS investiert werden musste. Mit einer größeren Besetzung hätte man zumindest noch die Parallelverarbeitung (jede Person arbeitet an Testprozeduren für ein anderes Teilsystem) erhöhen können, die ohnehin schon (mit regelmäßigen Arbeitsbesprechungen) betrieben wurde. Da aber in anderen Teilbereichen von TRACS auch niemand hätte abgezogen werden können, musste man diese Situation hinnehmen. Erst gegen Ende gab es noch einmal Unterstützung bei der Erstellung der Konzepte für die Integrationstests.

Die Ursachen für diese Situation sind sicher vielschichtig und können hier nicht vollständig geklärt werden, doch meinen wir, dass im Wesentlichen die zu späte Bereitstellung des Systems und der Spezifikation (wofür es wiederum zig verschiedene Ursachen gibt, eigentlich könnte man da fast einen Fehlerbaum aufstellen) sowie die über lange Zeit

vorliegende Vernachlässigung des Testbereiches zu nennen sind.

Marcin Dysarz, letztes Semester auch Arne Stahlbock

7.8 Simulator

7.8.1 Überblick

Im Rahmen des Projektes wurde ein Simulator für Straßenbahnnetze entwickelt. Dieser Simulator ist dabei so angelegt, dass mit ihm mehrere im Projektkontext stehende Aufgaben gelöst werden können, die sich wie folgt identifizieren lassen:

- Visualisierung des Gleisnetzes
  Der Simulator soll ein komplettes Gleisnetz im statischen Zustand visualisieren können, zur Überprüfung, ob das Gleisnetz in korrekter Form vorliegt. Dies dient zur Prüfung der Korrektheit des ebenfalls im Projekt TRACS entwickelten CAD-TND-Konverters.

- Visualisierung der Funktionsweise des Steuerinterpreters
  Um die Anweisungen des im Projekt entwickelten Steuerinterpreters anschaulich nachvollziehen und überprüfen zu können, wird ein Gleisnetz samt darauf fahrenden Straßenbahnen visualisiert und ein dynamischer Ablauf simuliert.

- Erstellung einer Testmöglichkeit
  Mit Hilfe des Simulators soll es möglich sein, bestimmte Situationen gezielt nachstellen zu können, um die Reaktion des Steuerinterpreters zur Laufzeit überprüfen zu können. Auch soll der Simulator für SI-Dauertests einsetzbar sein.

Allgemein gesagt, simuliert der Simulator ein Gleisnetz mit darauf fahrenden Straßenbahnen, sendet die Informationen über den jeweils aktuellen Zustand zum SI und empfängt und verarbeitet die Anweisungen, die der SI als Reaktion auf den aktuellen Zustand zurücksendet.

Dieser Abschnitt des Berichtes befasst sich mit der Konzeptionierung und Entwicklung dieses Simulators.

Niklas Polke, Arne Stahlbock

7.8.2 Anforderungen an den Simulator

Im Projekt TRACS ist das Ziel des Simulators eine zusätzliche (da nicht vollständige) und anschauliche Überprüfung der Funktionsweise und der Korrektheit des Steuerinterpreters (SI). Des Weiteren ergibt sich erst durch das Vorhandensein des Simulators die Möglichkeit für den SI, ein Gleisnetz (wenn auch nur simuliert) zu steuern. Mehr zur Funktion des SI in Kapitel 7.5 auf Seite 171.
Diese Ziele werden vom Simulator erfüllt, indem er ein beliebiges Gleisnetz – aufgeschrieben in einer Datei in TND-Form –, auf ihm fahrende Straßenbahnen sowie die

Zustände aller vorhandenen Hardwareteile visuell darstellt und deren Funktionsweise simuliert, die Anweisungen des SI befolgt und ihm Rückmeldung über den aktuellen Zustand des Systems gibt. Die Anforderungen an den Simulator sind die folgenden, die auch im weiteren Verlauf in Unterkapiteln näher erläutert werden:

- *Flexibilität* – Simulation verschiedenster Gleisnetze
  Der Simulator muss flexibel jedes beliebige Gleisnetz, das ihm in TND-Form übergeben wird, simulieren können. Dazu gehört das Einlesen der TND.
- *Berechnung der Simulation* der Funktionsweise der Hardwareelemente und der Straßenbahnen
  Was muss wie simuliert werden, damit die Simulation die Realität ausreichend genau beschreibt?
- *Visualisierung* der Zustände aller Hardwareelemente und der Straßenbahnen
  Was muss wie visualisiert werden, damit der Anwender des Simulators alle benötigten Informationen über den simulierten Zustand des Gleisnetzes bekommt?
- *Schnittstelle zum SI* – der Simulator simuliert das Verhalten eines Gleisnetzes (s. o.), das von einem SI gesteuert wird. Damit es gesteuert werden kann, müssen Treiber hergestellt werden. Diese müssen über die von der SI-Gruppe definierte Treiberschnittstelle ansprechbar sein.
- *Korrektheit*
  Die Korrektheit des Simulators wird im Rahmen des Projektes TRACS nicht von anderen Modulen verifiziert. Aus diesem Grund muss hier besonderen Wert auf eigene Überprüfungen der Korrektheit geachtet werden.
- *Zeitverhalten*
  Die Aktionen und Reaktionen in einem realen Gleisnetz laufen in bestimmten zeitlichen Größenordnungen ab. Damit die Simulation möglichst präzise die realen Bedingungen wiedergeben kann, muss darauf geachtet werden, dass die Simulation die realen Zeiten in etwa einhalten kann.
- *Systemumgebung*
  Es erscheint wünschenswert, den Simulator möglichst auf handelsüblichen PC-Systemen einsetzen zu können, zumal auch der SI als Kernprodukt des Projektes auf Standard-PCs funktionieren soll.

7.8.2.1 Flexibilität – Simulation verschiedenster Gleisnetze

Der Steuerinterpreter (SI) des Projektes TRACS wird aus den Informationen über das Gleisnetz – welche in Form einer Tram Network Description (TND) vorliegen – generiert.

Damit der Simulator die gleiche Menge an Gleisnetzen beherrscht und damit die Funktionsweise des SI in allen Fällen darstellen sowie möglichst vielseitig testen kann, muss auch er die Gleisnetze in TND-Form einlesen können. Dazu muss er folgende Blöcke der TND einlesen und interpretieren können:

- *Definitions*-Block
  Dieser Block enthält die Informationen, welche Hardwareelemente im Gleisnetz vorhanden sind, welchen Namen (ID) sie haben und von welchem Typ sie sind.

- *Coordinates*-Block
  Unabdingbar für eine sinnvolle Darstellung des Gleisnetzes sind die Informationen über die Standorte der Hardwareelemente – jegliche Bewegung der Straßenbahnen benötigt die Informationen über Ortskoordinaten und Entfernungen im Gleisnetz.

- *Signal Properties*-Block
  In diesem Block sind funktionale Eigenschaften der im Gleisnetz vorkommenden Signale aufgeführt.

- *Point Properties*-Block
  In diesem Block sind funktionale Eigenschaften der im Gleisnetz vorkommenden Weichen aufgeführt.

- *Slipswitches*-Block
  Hier werden diejenigen Kreuzungen und Weichen genannt, die zusammen Kreuzungsweichen (slipswitches) bilden.

- *Relations*-Block
  Wie die Weichen, Kreuzungen und Sensoren miteinander verbunden sind bzw. zwischen welchen Elementen sich Gleisstücke befinden, das steht in diesem Block. Auch werden hier Endpunkte des Gleisnetzes definiert.

- *SigPositions*-Block
  Die Zuordnung eines Signals zu einem Sensor bzw. einem Standort zwischen zwei Sensoren erfolgt in diesem Block.

- *RouteDefinitions*-Block
  In diesem Block steht, welche Route über welche Sensoren führt. Diese Information wird benötigt, um während der Simulation entscheiden zu können, ob eine simulierte Straßenbahn (die eine bestimmte Route fahren will) vom SI tatsächlich über die verlangte Route geleitet wird.

- *Clearances*-Block
  Hier sind die für eine Route nötigen Signalstellungen aufgeführt. Eine simulierte Bahn soll nur fahren, wenn ihre gewünschte Richtung freigegeben ist.

Diese Informationen muss der Simulator speichern, um während einer Simulation darauf zurückgreifen zu können. Um eine flüssige und möglichst genaue Simulation gewährleisten zu können, sollte die Datenstruktur so aufgebaut sein, dass ein Auslesen (eventuell verbunden mit einem Suchen) sehr effizient vorgenommen werden kann.

7.8.2.2 Berechnung der Simulation

Die Simulation ist die Kernaufgabe des Simulators. Möglichst realistisch sollen in diesem Simulator die Reaktionsweisen von Sensoren, Signalen und Weichen simuliert werden. Damit diese Hardwareelemente überhaupt auslösen, brauchen sie Straßenbahnen, auf die sie reagieren können – diese Straßenbahnen müssen folglich auch simuliert werden. Zwei grundlegende Fälle sollen simuliert werden können:

- korrekt funktionierendes Gleisnetz
  In diesem Fall wird davon ausgegangen, dass das gesamte Gleisnetz korrekt funktioniert. Das bedeutet, dass alle Hardwareelemente gemäß ihrer Parameter wie gewünscht funktionieren und die Straßenbahnen so fahren, wie man es von einem ‚normalen' Straßenbahnfahrer erwartet.

- (zum Teil) fehlerhaft agierende Elemente
  In diesem Fall soll getestet werden, wie der Steuerinterpreter auf unvorhergesehene Ereignisse reagiert – dazu gehören Hardwareelemente, die nicht reagieren oder die fehlerhafte Informationen liefern. Auch können Bahnen unvermittelt stehen bleiben oder Signale missachten.

7.8.2.2.1 Korrekt funktionierendes Gleisnetz Das gesamte zu simulierende System besteht also aus einem zunächst statischen Gleisnetz (zusammengesetzt aus Gleisen, Sensoren, Signalen und Weichen), welches durch Hinzufügen von Straßenbahnen, die an einer Einfahrt in das Gleisnetz eintreten, zu einem dynamischen System wird, in dem folgende Interaktionen simuliert werden müssen:

- Reaktion: Signal $\Longrightarrow$ Bahn
  - Eine Bahn hält an, wenn sie rechtzeitig ein für ihr Gleis zuständiges Signal ‚sieht', das ihr die Weiterfahrt in die gewünschte Richtung verwehrt.
  - Eine Bahn hält nicht an, wenn sie ein für ihr Gleis zuständiges Signal auf Stop springen ‚sieht', aber schon so nah ist, dass ein Anhalten davor nicht mehr möglich ist.
  - Eine Bahn fährt durch, wenn sie ein für ihr Gleis zuständiges Signal ‚sieht', das ihr die Weiterfahrt in die gewünschte Richtung erlaubt.

- Reaktion: Weiche $\Longrightarrow$ Bahn

  - Wenn eine Bahn von der spitzen Seite auf eine Weiche gelangt, die sich nicht im undefinierten Schaltzustand befindet, fährt die Bahn auf der gerade geschalteten stumpfen Seite wieder aus.
  - Fährt eine Bahn von einem stumpfen Ende auf eine Weiche, die auffahrbar ist oder die gerade auf dieses stumpfe Ende geschaltet ist, gelangt die Bahn an das spitze Ende.
  - Fährt eine Bahn von der spitzen Seite auf eine sich in einem undefinierten Schaltzustand befindliche Weiche, schaltet eine Weiche gerade um, während eine Bahn sich auf ihr befindet oder fährt eine Bahn von einer nicht geschalteten stumpfen Seite auf eine nicht auffahrbare Weiche, so kommt es zum Schadensfall.

- Reaktion: Bahn $\Longrightarrow$ Sensor
  - Sensoren bemerken, wenn eine Bahn in ihren Einflussbereich ein- bzw. austritt und ändern – je nach Funktionsweise – ihren eigenen Zustand.
  - RouteRequestSensoren erhalten den Routenwunsch von Bahnen, wenn diese sich neben/auf ihnen befinden.

- Reaktion: Bahn $\Longrightarrow$ Weiche
  - Wenn eine Bahn über eine Weiche von der Seite kommt, auf der mehrere Gleise abzweigen, so schaltet die Weiche um, so dass sie anschließend auf die Verzweigung eingestellt ist, über die gerade die Bahn gefahren ist, wenn die Weiche auffahrbar ist. Handelt es sich um eine Rückfallweiche, schaltet sie anschließend wieder in den vorigen Zustand zurück.

- Reaktion: Bahn $\Longrightarrow$ Bahn
  - Eine Bahn erkennt den Fall, wenn sie auf eine ihr voraus fahrende Bahn auffahren würde und kommt vor einer Berührung hinter der anderen Bahn zum stehen. Sie fährt anschließend weiter, sofern die voraus fahrende Bahn weiterfährt. Damit soll das Einhalten von Sicherheitsabständen durch die Fahrer simuliert werden. (Das Nichteinhalten könnte nicht durch den SI verhindert werden, daher muss dieser Fall nicht berücksichtigt werden.)
  - Fährt eine Bahn mit dem Kopf voraus in den Kopf einer anderen Bahn, so wird ein Crash erkannt.
  - Fährt eine Bahn einer anderen auf einer Weiche oder auf einer Kreuzung in die Seite hinein, so wird ein Crash erkannt, denn dieser Fall muss nicht vom Fahrer der Straßenbahn vorhergesehen, sondern soll vom Steuerinterpreter verhindert werden.

  - Eine Bahn fährt mit gesetzter Route los, sobald sie an einer Einfahrt auf das Gleisnetz gelangt.
  - Eine Bahn fährt immer mit konstanter Geschwindigkeit (Ausnahme: Anhalten an Signalen oder hinter einer Bahn). Variable Geschwindigkeiten oder Halt an Haltestellen werden nicht vorgesehen, da letztlich der Steuerinterpreter nur eine Folge von Sensorenauslösungen als Input bekommt. Wichtig ist die Reihenfolge der Auslösungen, nicht die dazwischen liegenden Zeitabstände (zumal diese durch unterschiedliche Verkehrsverhältnisse, Witterungsbedingungen etc. beeinflusst werden können, so dass der SI hier mit allen möglichen Abständen umgehen muss).

- Reaktion: Sensor $\Longrightarrow$ Bahn und Bahn $\Longrightarrow$ Signal
  - Für beide Fälle sind keine Reaktionen vorgesehen, da auch in der Realität keine hier stattfindende Reaktion erkennbar ist.

Die Einstellungen, welche Route eine Straßenbahn befährt und wie schnell sie sich fortbewegen kann, sollen vom Anwender vor dem Start gesetzt werden können. Ebenso soll es möglich sein, initial Weichen und Signale zu stellen, damit gezielt getestet werden kann, wie der Steuerinterpreter auf die gestellte Situation reagiert. Beeinflussungen durch den Nutzer im laufenden Betrieb sind nicht explizit vorgesehen.

7.8.2.2.2 Fehlerhaft agierendes Gleisnetz Mit Hilfe eines simulierten fehlerhaft agierenden Gleisnetz soll getestet werden, wie der Steuerinterpreter z. B. mit Ausfällen von Hardwareelementen fertig wird bzw. wie er auf diese reagiert. Folgende fehlerhafte Situationen sollen dafür simuliert werden können:

- Ausfall eines Hardwareelements (Sensor, Signal, Weiche), d.h. keine Reaktion des Elementes mehr
- Eigenmächtige Schaltvorgänge von Weichen und Signalen bzw. Sensorauslösungen, wenn keine Bahn am Sensor vorbeigekommen ist (allgemein: nicht erwartete Vorgänge)
- Nichtbeachten von Signalen durch Bahnen (zwar kann der SI in diesem Fall einen Unfall nicht verhindern, jedoch müsste er Vorkehrungen treffen, den Schaden gering zu halten, z.B. durch Absperren des betroffenen Abschnittes).

7.8.2.3 Visualisierung

Der Simulator ist das einzige Modul des Projektes TRACS, welches die Funktionsweise des Steuerinterpreters *anschaulich* darstellt. Aus diesem Grund ist die Visualisierung

eine wichtige Anforderung. Damit die Funktionsweise leicht nachvollzogen werden kann, müssen folgende Aspekte visualisiert werden:

- Gleise
  Der Aufbau des gesamten Gleisnetzes soll möglichst genau dargestellt werden. Wichtig hierbei ist, dass die Streckenlängen möglichst präzise sind und der Gesamteindruck des Gleisnetzes stimmt. Es muss außerdem ganz deutlich sein, an welchen Stellen sich die Strecken kreuzen. Die genaue Form von Kurven ist für die Simulation unwichtig.

- Sensoren (Sensors)
  In der Visualisierung des gesamten Gleisnetzes sollen die Positionen der Sensoren, der Typ und der Zustand der z. T. unterschiedlich agierenden Sensoren erkennbar sein. Diese Informationen sollen möglichst deutlich dargestellt werden, damit der Anwender leicht nachvollziehen kann, was der Sensor zum Zeitpunkt gerade tut.

- Weichen (Points)
  Die Darstellung der Weichen soll erkennen lassen, in welchem Zustand sich gerade die Weiche befindet – also ob sie zum Beispiel gerade nach links oder nach rechts geschaltet ist, oder ob sie sich gerade in einem Umschaltvorgang befindet. Der Typ der Weiche soll außerdem erkennbar sein.

- Signale (Signals)
  Für die Visualisierung eines Signals ist es wichtig, sowohl die Anzeige als auch die Richtung des Signals zu verdeutlichen. Wenn z. B. auf einer Geraden Straßenbahnen sowohl in die eine, als auch in die andere Richtung fahren, dann ist es wichtig als Anwender des Simulators zu erkennen, ob das Signal, an dem die Bahn gerade vorbeifährt, für die Weiterfahrt der Bahn zuständig ist.

- Straßenbahnen (Trams)
  Die Darstellung einer fahrenden Bahn sollte erkennen lassen, wo sich der Kopf und wo sich das Ende befinden (also in welche Richtung die Straßenbahn fährt) und auf welchem Bereich auf dem Gleisnetz die Bahn gerade steht. Dies ist wichtig für die manuelle Kontrolle, ob die Sensoren richtig funktionieren, ob die Bahn auf die Signale reagiert und ob ein Zusammenstoß auch richtig erkannt wird.

Da die Hauptaufgabe der Visualisierung die Darstellung des Systemzustands ist – also die Komposition der Zustände der einzelnen Sensoren, Signale, Weichen und Bahn(en) – spielt die Formschönheit lediglich eine untergeordnete Rolle.
Die Darstellung soll grafisch am Bildschirm erfolgen.
Ferner muss die Darstellung geeignet sein, die Korrektheit der CAD-TND-Umwandlung (Näheres zum CAD-TND-Konverter in den Kapiteln 7.2.3.2 auf Seite 89 und 7.2.4 auf Seite 104) zu verifizieren. Dazu müssen also alle Details, die in der DXF-Darstellung vorkommen, auch in der Visualisierung des Simulators ihren Platz finden.

7.8.2.4 Schnittstelle zum SI

Der Simulator simuliert lediglich die Reaktionen des Gleisnetzes auf die Anweisungen eines Steuerinterpreters (SI) sowie die Straßenbahnen. Ohne SI würden folglich die Meldungen der Sensoren unbeachtet bleiben, und die Signale würden überhaupt nicht und die Weichen nur passiv geschaltet werden. Da der SI nicht für die Steuerung eines Simulators entwickelt wird, benötigt der Simulator eine Schnittstelle, um mit dem SI zu kommunizieren. Der SI verfügt über eine Treiberschnittstelle, an der Hardware-Treiber für Weichen, Sensoren und Signale angebunden werden können. Für den Simulator sind daher Treiber zu entwickeln, die diese Schnittstelle nutzen und die für den SI wie „normale Hardwaretreiber“ erscheinen.
Damit die Komposition aus Simulator und SI funktionieren kann, muss eine bidirektionale Kommunikation stattfinden. Der SI muss vom Simulator über dessen aktuellen Zustand benachrichtigt werden, wenn dieser seinen Zustand ändert und der Simulator muss seinerseits vom SI benachrichtigt werden, wenn dieser die Umstellung eines Signals oder einer Weiche befiehlt. Die zu realisierenden Aspekte der Kommunikation sind im Folgenden aufgelistet und beschrieben:

- Reaktion: Sensoren $\Longrightarrow$ SI
  - Der SI muss permanent über die aktuellen Zustände der Sensoren informiert sein.
- Interaktion: Signale $\Longleftrightarrow$ SI
  - Der SI muss permanent über die aktuellen Zustände der Signale informiert sein.
  - Die Signale im Simulator befolgen die Anweisungen vom SI.
- Interaktion: Weichen $\Longleftrightarrow$ SI
  - Der SI muss permanent über die aktuellen Zustände der Weichen informiert sein.
  - Die Weichen im Simulator befolgen die Anweisungen vom SI.

Da die Kommunikation bidirektional ist und eine Menge Daten praktisch gleichzeitig hin- und hergeschickt werden müssen, ist eine effiziente und präzise formulierte Schnittstelle von Nöten. Die Datenpakete sollten dafür möglichst klein sein.

7.8.2.5 Korrektheit

Der im Projekt TRACS zu entwickelnde Simulator ist kein zentrales Modul des Projektes und wird nicht in die Kette von Verifikationen im Projekt eingebunden. Bei der

Entwicklung muss folglich darauf geachtet werden, dass dieser korrekt funktioniert.
Ziel des Simulators ist die Möglichkeit, den Steuerinterpreter anschaulich testen zu können. Wenn bei diesem Testen Fehler auftreten, gibt es beim Zusammenspiel von Steuerinterpreter, Treibern und Simulator folglich drei Fehlerquellen.
Die Treiber sollen möglichst simpel gehalten werden und es besteht nach Meinung der Autoren eine sehr hohe Wahrscheinlichkeit, dass diese entweder permanent deutliche Fehler produzieren oder korrekt funktionieren. Bevor man im Weiteren auf einen Fehler im SI sucht, sollte auch die Fehlerquelle Simulator zuvor überprüft werden.
Um im Voraus die Fehlerwahrscheinlichkeit beim Simulator so gering wie möglich zu halten, sollten folgende Aspekte beachtet und befolgt werden:

- Übersichtliche Struktur
  Eine übersichtliche, gut durchdachte, möglichst einfache und selbsterklärende Struktur kann einerseits im Voraus die Fehlerquellen deutlich vermindern und andererseits die Implementierung erleichtern, was wiederum indirekt weniger Fehler verursacht. Auch im Falle eines Tausches von Projektteilnehmern erleichtert eine gute Struktur diesen den Einstieg und mindert auch hier die Fehlerursachen.

- Gründliche Tests
  Die einzelnen Module und Funktionen/Methoden des Simulators sollen so gründlich durch Tests auf ihre Funktionsweise hin überprüft werden, dass Fehler möglichst ausgeschlossen werden können. Ziel dieser Tests ist jedoch nicht eine Verifikation des Simulators, welche eine korrekte Funktionsweise beweisen würde. Im Wesentlichen soll sichergestellt werden, dass die oben im Abschnitt „Berechnung der Simulation“ (7.8.2.2) beschriebenen Reaktionen korrekt ablaufen, dass die grafische Darstellung funktioniert und dass die Treiber zur Anbindung an den SI wie vorgeschrieben reagieren.

Wie genau diese Anforderungen auch umgesetzt werden, es kann bei der Auswertung von auftretenden Fehlern bei der Anwendung des Simulators in Kombination mit dem Steuerinterpreter nie ausgeschlossen werden, dass der Fehler beim Simulator liegt. Der Simulator stellt jedoch nur eine untergeordnete Testmöglichkeit für den SI dar, ist daher nicht sicherheitskritisch und muss daher nicht formal verifiziert werden.

7.8.2.6 Zeitverhalten

Das ganze System, welches letztendlich zum Laufen gebracht werden soll, besteht aus einem Steuerinterpreter, einem Simulator und den für die Kommunikation benötigten Treibern. Simuliert werden soll also ein System – bestehend aus einem Gleisnetz samt fahrender Straßenbahnen –, das in einer solchen Zeit läuft, die möglichst realitätsnah bezüglich Fahrgeschwindigkeiten, Schaltzeiten von Weichen und Signalen sowie Auslösezeiten von Sensoren ist. Von einem Computer mit nur einem Prozessor (der keine zwei

Anweisungen tatsächlich parallel ausführen kann) ist Echtzeit nicht realisierbar. Dennoch soll der Simulator mit einem Computer mit nur einem Prozessor auskommen, weswegen das Zeitverhalten in der Realität vom Simulator nicht exakt eingehalten werden kann, aber möglichst naturgetreu wiedergegeben werden können soll.
Folgende Anforderungen definieren die Bedingungen, die erreicht werden müssen:

- Sensoren, Weichen und Signale sollen so schnell reagieren und ihre Meldungen übertragen, wie es in der Realität der Fall ist. Da im Simulator keine analogen Meldungen verschickt und entschlüsselt werden müssen, lässt sich hier bereits ein wenig Zeit gewinnen.
- Die Übertragungszeit einer Anweisung vom Steuerinterpreter zu einem Signal oder einer Weiche soll der realen Übertragungszeit möglichst ähnlich sein.
- Die simulierten Straßenbahnen sollen ihre Entscheidungen möglichst in Echtzeit treffen. Die Reaktionszeit einer Straßenbahn, die kurzfristig noch ein rotes Signal bekommt, soll möglichst realistisch simuliert werden.

Dieser Aspekt sollte bei allen Aufgaben zur Realisierung dieses Simulators gut im Auge behalten werden, damit gleich im Voraus bei der Struktur des Simulators keine unnötigen Erschwernisse eingebaut werden. Ein Verzicht auf Speicheroptimierung zugunsten der Geschwindigkeit bietet sich hier an.
Auf unterschiedliche Simulationsgeschwindigkeiten wird verzichtet, da der SI auch im realen Leben nicht mit Elementen, die in völlig anderen zeitlichen Größenordnungen arbeiten, konfrontiert würde.

7.8.2.7 Systemumgebung

Die Systemumgebung, in der der Simulator laufen soll, muss bestimmten Anforderungen gerecht werden.
So soll der Simulator genau wie den SI auf Standard-PC-Hardware laufen können. Auch an das Betriebssystem sollen keine besonderen Forderungen gestellt werden. Vorgesehen ist, ein gewöhnliches (d.h. nicht speziell konfiguriertes oder gar umprogrammiertes) Linux einzusetzen. Sollte der Simulator zusätzlich auf weiteren Systemen laufen, so wäre das ein Nebeneffekt, auf den nicht gezielt hingearbeitet wird, der aber auch durchaus zu begrüßen wäre.
Das System, auf dem der Simulator läuft, muss mit demjenigen, auf dem der SI läuft, kommunizieren können. Da nicht davon auszugehen ist, dass beide zeitgleich auf demselben Rechner laufen (da der Betrieb des Simulators sonst unerwünschte Auswirkungen auf den Betrieb des SI haben könnte) – lediglich die Treiber zur Anbindung des Simulators müssen auf dem SI-Rechner laufen –, muss eine Netzwerkverbindung zwischen Treiber und „Rest des Simulators“ vorgesehen werden.

7.8.3 Weltmodell und Abstraktionen

Für den Simulator ist selbstredend das projektweite Weltmodell, beschrieben in Kapitel 2 auf Seite 10, insbesondere der Abschnitt zur Bahntechnik in 2.3 auf Seite 18 bindend. An dessen Entwicklung hat sich die Simulatorgruppe maßgeblich beteiligt.
An einigen Stellen werden jedoch für den Simulator Vereinfachungen und Abstraktionen vorgenommen, die in besagtem Kapitel nicht erscheinen und die daher hier beschrieben werden sollen:

- Jede Bahn fährt mit einer konstanten Geschwindigkeit (diese Geschwindigkeit kann allerdings von Bahn zu Bahn unterschiedlich sein). Ausnahmen sind das Halten vor einem Signal oder das Fahren hinter einer langsameren Bahn. Zu begründen ist diese Vereinfachung damit, dass die dem SI gelieferten Daten über den Fahrtverlauf von Bahnen sich auf Abfolgen von Sensorauslösungen beschränken und Bahnen, die ihre Fahrgeschwindigkeiten ändern, nur die Abstände zwischen den Auslösungen verändern, nicht aber die Folge an sich. In diesem Zusammenhang werden im Simulator daher auch keine, die Geschwindigkeit beeinflussende Umstände berücksichtigt (Haltestellen, andere Verkehrsteilnehmer, Fahrgäste, geographische Gegebenheiten, Witterung, ...).

- In der Simulation wird davon ausgegangen, dass der Abstand, in dem ein Signal gesehen werden kann, immer gleich ist. Damit wird eine Abstraktion für das Fahrerverhalten bei unterschiedlichen Sichtweiten vorgenommen. Kann der Fahrer in Realität ein Signal erst spät einsehen, verringert sich sein Reaktionszeitfenster, was er durch langsamere Fahrt wieder vergrößern müsste. Im Endeffekt ist damit das Reaktionszeitfenster in beiden Fällen etwa gleich groß, was mittels obiger Annahme simuliert werden kann.

- Das Fahrerverhalten in der Simulation wird durch zwei Regeln beschrieben: Signale, die auf Stop schalten, werden beachtet, wenn die Bahn noch einen bestimmten Mindestabstand zum Signal aufweist (die vorgenannte Sichtweite), andrenfalls fährt die Bahn durch. Auf vorherfahrende Bahnen wird nicht aufgefahren. Das Beachten von Signalen durch den Fahrer kann allerdings auch abgeschaltet werden, wenn solche Situationen simuliert werden sollen.

- Weichen, Sensoren und Signale können ausfallen oder Fehlverhalten aufweisen, es ist jedoch irrelevant, durch welche speziellen Umstände dieses Fehlverhalten bewirkt wird (Stromausfälle, Hardwaredefekte, Verschleiß, ...). Daher werden auch solche Umstände nicht berücksichtigt, sondern lediglich besagte Ausfälle und Fehlverhalten simuliert, ohne auf nähere Ursachen einzugehen.

- Analog gilt dies auch für Ausfälle von Bahnen.

- Entgleisungen von Bahnen kann der SI nicht verhindern, daher werden sie und die sie auslösenden Gegebenheiten auch nicht simuliert. Ausnahmen hiervon sind das Umschalten einer Weiche, während eine Bahn anwesend ist, oder das Auffahren auf eine nicht auffahrbare Weiche – diese werden in der Simulation berücksichtigt.

Arne Stahlbock

7.8.4 Konzept des Simulators

Das Konzept beschreibt eine abstrakte Struktur des Simulators, die die Anforderungen auf Basis des Weltmodells umsetzt und realisiert. Es dient anschließend als Grundlage für die Implementierung. Das Konzept ist zusammengesetzt aus folgenden Teilen:

- Programmiersprache
  Wir benutzen im Wesentlichen die Programmiersprache „Java™“ – in diesem Unterkapitel wird aufgezeigt, warum wir uns für diese Sprache entschieden haben.
- Programmstruktur
  Die Programmstruktur erläutert die Klassenstruktur des Programms und gibt einen Überblick über die Bedeutung der einzelnen Klassen.
- Flexibilität
  In diesem Unterkapitel wird beschrieben, wie der Simulator Gleisnetze in TND-Form einliest und damit flexibel jedes beliebige Gleisnetz verarbeiten kann.
- Simulation
  Dieses Unterkapitel umfasst den Kern des Simulators – die Abläufe und die Methoden zur Berechnung des Simulationszustands werden beschrieben.
- Visualisierung
  An dieser Stelle werden die Konzepte zur Visualisierung des Gleisnetzes samt Straßenbahnen aufgezeigt.
- Treiber zur Anbindung an den SI
  In diesem Unterkapitel wird die Schnittstelle zwischen Simulator und den Treibern, die der SI zur Steuerung der Simulation benutzt – Ablauf und Format der Nachrichten – beschrieben.
- Korrektheit
  Konzepte zur Sicherstellung der Korrektheit des Simulators werden an dieser Stelle aufgezeigt und erläutert.
- Echtzeitfähigkeit
  Die Struktur, welche dazu führt, dass der Simulator ein Echtzeitsystem sehr präzise simuliert, wird hier näher beleuchtet.

- Hardware
  Zuletzt wird die Zielhardware beschrieben, für die der Simulator entwickelt wird.

7.8.4.1 Programmiersprache

Zur Programmierung eines Simulators mit vielen gleichen Elementen bietet sich das Konzept der objektorientierten Programmierung (OOP) an. OOP geht von der Grundlage aus, dass Objekte programmiert werden, die sowohl Daten als auch Algorithmen enthalten. Das Gegenteil findet man in der prozeduralen Programmierung, in der diese beiden Elemente getrennt von einander programmiert werden. Einige verbreitete OOP-Sprachen sind Java™, C++, Simula und Smalltalk, von denen die letzten beiden aber wegen mangelnder Kenntnisse der Simulatorgruppe über diese Sprachen von vornherein ausgeschlossen waren. Um die Entscheidung zu treffen, ob nun Java™oder C++ verwendet werden soll, werden die Vor- und Nachteile beider Sprachen gegenüber gestellt, die Schnittstellen zu anderen Arbeitsgruppen betrachtet und die persönlichen Erfahrungen der Mitglieder der Arbeitsgruppe Simulator miteinbezogen.

7.8.4.1.1 Vor- und Nachteile von Java™gegenüber C++ Jede Programmiersprache hat ihre Vorteile und Nachteile. Anhand der wichtigsten Unterschiede zwischen den beiden Sprachen Java™und C++ soll untersucht werden, welche Sprache sich in Hinsicht auf unseren Simulator besser eignet.
Java™wurde auf der Grundlage der Sprachen C und C++ mit dem Ziel entwickelt, die Vorteile beider Sprachen zu übernehmen und die Nachteile dieser Sprachen möglichst zu vermeiden. Dadurch entstanden u. a. Unterschiede in der Speicherverwaltung, Portabilität, Grafikprogrammierung, Rechenzeit und Vererbung. Die genannten Unterschiede werden jetzt genauer untersucht.

- Vorteile von Java™gegenüber C++
  Unter dem Gesichtspunkt, einen Simulator mit grafischer Oberfläche zu programmieren, bietet Java™in den Bereichen Speicherverwaltung, Portabilität und Grafikprogrammierung Vorteile gegenüber C++.

  – Speicherverwaltung
    In C++ muss für Datenstrukturen explizit Speicher reserviert und auch wieder freigegeben werden. Dadurch treten besonders leicht Programmierfehler auf. In Java™muss man sich nicht darum kümmern; bei der Erstellung eines Objekts wird automatisch genügend Speicher zur Verfügung gestellt. Außerdem muss man sich nicht um die Freigabe von nicht mehr genutztem Speicher kümmern, da es in Java™einen sog. *Garbage-Collector* gibt, der diese Aufgabe automatisch erledigt. Also gibt es in Java™eine Möglichkeit weniger, Programmierfehler zu machen (vgl. [Krü03, S. 39]).

  - Portabilität
    Java™stellt im Vergleich zu C++ standardmäßig eine weitaus größere Klassenbibliothek zur Verfügung, die die gleichen Funktionen auf allen Plattformen verwirklicht. Z. B. sind Netzwerkprogrammierung, Serialisierung und Grafikprogrammierung in der Basis-Version von Java™verfügbar. Diese drei Elemente waren auch für den Einsatz in unserem Simulator geplant (auf Serialisierung wurde dann später verzichtet). Ein weiterer Unterschied ist die Portabilität des kompilierten Quellcodes in Java™. Der Quellcode wird in Bytecode übersetzt, welcher von einer Java™Virtual Machine auf einer beliebigen Plattform interpretiert werden kann. Im Gegensatz dazu, ist es bei C++-Programmen notwendig, den Code für eine neue Plattform neu zu kompilieren, da der kompilierte C++-Code direkt auf die benutzte Plattform ‚zugeschnitten' wird (vgl. beide Unterschiede mit [Krü03, S. 45]).
  - Grafikprogrammierung
    Da die Grafikprogrammierung einen recht großen Teil in unserem Simulator darstellt, sollte man sich mit diesem Thema noch genauer auseinander setzen. Wie oben angesprochen, steht „mit Java™und ihren Klassenbibliotheken [...] erstmals eine [...] Sprache zur Verfügung, die das Erstellen von GUI-Programmen bereits als Kernfunktionalität bietet" ([Krü03, S. 41]). Neben elementaren Grafikfunktionen stellt Java™das *Swing Toolset* bereit, das viele Funktionen enthält, um grafische Elemente zur Interaktion mit dem Benutzer zu erstellen und anzupassen.

- Nachteile von Java™gegenüber C++
  Für unsere Aufgabe, einen Simulator zu programmieren, sind die Nachteile von Java™in den Bereichen Rechenzeit und Vererbung von Interesse und sollten somit genauer untersucht werden.

  - Rechenzeit
    Da Java™-kompilierter Code in einem Byte-Format vorliegt, muss dieser zur Laufzeit erst noch von einer Java™Virtual Machine vorinterpretiert werden, bevor der Prozessor die Befehle ausführen kann. Dadurch erhöht sich die Rechenzeit. In C++ ist der kompilierte Code direkt vom Prozessor les- und ausführbar. Dieser Zeit-Unterschied macht sich jedoch nur bei rechenintensiven Programmen bemerkbar (vgl. [Krü03, S. 45]). Die einzigen rechenintensiven Leistungen in unserem Simulator werden Such-Operationen sein, die jedoch bei der Initialisierung durchgeführt und zur Laufzeit vermieden werden. Aus diesem Grund ist Java™für unsere Bedürfnisse ausreichend schnell.
  - Vererbung
    In Java™ist keine Mehrfachvererbung wie in C++ möglich, nur eine linea-

re Einfachvererbung kann erreicht werden. Um trotzdem die Möglichkeit zu bieten, dass einige Klassen die gleichen Methoden erben/implementieren, wurden in Java™die *Interfaces* eingeführt (vgl. [Krü03, S. 38]). Da in unserem Simulator verschiedene Hardware-Elemente simuliert werden sollen, die recht viele Gemeinsamkeiten untereinander haben, ist eine Mehrfachvererbung wünschenswert. Jedoch kann auch mit Einfachvererbung und Interfaces das gleiche Ergebnis erzielt werden.

- Fazit
  Es wird festgestellt, dass die genannten Vorteile von Java™bei der Programmierung des Simulators zum Tragen kommen und die Nachteile von Java™vernachlässigbar sind. Somit ist Java™die geeignetere Wahl für unseren Simulator.

7.8.4.1.2 Schnittstellen des Simulators Der Simulator hat lediglich zwei Schnittstellen zu Arbeitspaketen anderer Gruppen. Die erste Schnittstelle ist die im Projekt TRACS entwickelte Beschreibungssprache TND, die andere ist die Kommunikation mit dem Steuerinterpreter.

- Schnittstelle TND
  Das Gleisnetz, welches simuliert werden soll, liegt im TND-Format vor. Unser TRACS-Compiler, der dieses Format einliest und für die anderen Arbeitsgruppen weiter verarbeitet, ist in C geschrieben – er könnte also nur direkt Anwendung im Simulator finden, wenn dieser C++ verwenden würde. Würde der Simulator aber in Java™geschrieben werden, so gibt es zwei Möglichkeiten, um das TND-Format einzulesen.
  Die erste Möglichkeit wäre, einen zusätzlichen Compiler zu schreiben, der den Output des TRACS-Compilers von C in Java™übersetzt. Dies würde viel Programmieraufwand bedeuten, jedoch wäre der größte Aufwand nicht die Implementierung, sondern die Verifikation dieses Compilers. Aus diesem Grund kommt ein Compiler C→Java™nicht in Frage.
  Der andere Weg ist, einen eigenen Compiler in Java™zu programmieren, der das TND-Format einliest und eine Klassenstruktur für den Simulator erstellt. Dies ist nicht sonderlich schwierig, bedeutet jedoch etwas mehr Aufwand. Siehe hierzu den kommenden Abschnitt 7.8.4.3.

- Schnittstelle Steuerinterpreter
  Der Simulator kommuniziert während der Laufzeit mit dem Steuerinterpreter. Die Kommunikation erfolgt hierbei mittels Shared Memory, das auf dem Rechner, auf dem der SI läuft, angelegt wird. Um diesen Rechner wiederum an denjenigen, auf dem der Simulator läuft, anzukoppeln, wird eine Netzwerkverbindung auf TCP/IP-Basis benutzt, über die Meldungen in Form von Strings, also Zeichenketten versendet werden sollen. Für die Umsetzung der Informationen, die der SI

in das Shared Memory schreibt, in die an den Simulator zu versendenden Strings – und entsprechend für die Gegenrichtung – werden Treiber benötigt, die den Anforderungen des SI an Hardware-Treiber genügen. Diese Treiber werden in C geschrieben, um an den ebenfalls in dieser Sprache geschriebenen SI angebunden werden zu können. Texte können sowohl von C als auch von Java™gleich interpretiert werden, d. h. die Programmiersprache des eigentlichen Simulators kann unabhängig von der der Treiber (C) gewählt werden.
Die Kommunikation mit dem Steuerinterpreter und die simulatorinterne Kommunikation Treiber - eigentlicher Simulator wird später im Abschnitt 7.8.4.6 genauer beschrieben.

- Fazit
  Daraus ergibt sich, dass für das Einlesen des Gleisnetzes im TND-Format die Programmiersprache C++ von Vorteil wäre. Die Benutzung der Sprache Java™würde mehr Aufwand bedeuten, ist aber auch möglich. Für die Kommunikation mit den Treibern ist es egal, ob Java™oder C++ benutzt wird.

7.8.4.1.3 Persönliche Erfahrungen Die derzeitigen Mitglieder der Arbeitsgruppe Simulator haben sowohl Erfahrungen mit der Programmiersprache Java™als auch mit C++, wobei die Java™-Kenntnisse deutlich überwiegen. Des Weiteren wurde die Programmierung einer graphischen Benutzeroberfläche bisher nur mittels Java™in Erfahrung gebracht. Somit sind alle Mitglieder ‚fitter' in Java™als in C++, was bedeutet, dass wir in Java™weniger Einarbeitungsaufwand benötigen und effektiver programmieren können.

7.8.4.1.4 Fazit Es hat sich herausgestellt, dass Java™für unsere Zwecke die geeignetere Sprache ist. Auch die persönlichen Erfahrungen sprechen für Java™. Nur beim Einlesen des Gleisnetzes wäre die Wahl von C++ von Vorteil. Wir gewichten die Vorteile von Java™gegenüber C++ schwerer, da die Effizienz beim Java™-Programmieren größer sein wird. Der Nachteil, der durch den Zusatzaufwand eines Compilers, der in Java™geschrieben werden muss, entsteht, kann durch die schnellere und sichere Programmierung in Java™ausgeglichen werden. Demnach wird zwar mehr Quellcode produziert, jedoch wird nicht mehr Zeit benötigt.

7.8.4.2 Programmstruktur

Die Programmstruktur gibt einen Überblick über die Klassen, die programmiert werden, wie diese Klassen voneinander abhängen und welche Gemeinsamkeiten sie verbinden. Der gesamte Simulator gliedert sich in die beiden Pakete

- sim

- gui

Das Paket „sim“ enthält alle Berechnungen und (Kommunikations-)Abläufe, die der Simulator durchführt. Im Paket „gui“ sind die kompletten Grafikroutinen ausgelagert, die zur Visualisierung der Simulation dienen.

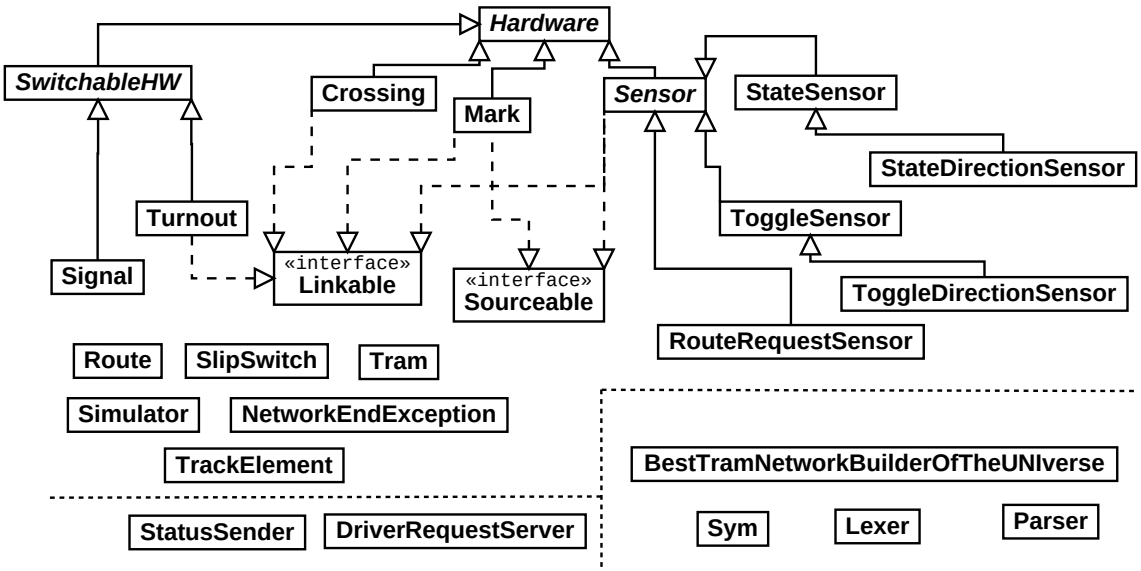


Abbildung 7.62: Klassendiagramm: Package ‚sim‘

7.8.4.2.1 Paket „sim“ In Abbildung 7.62 sind alle Klassen enthalten, die zur Simulation eines Gleisnetzes, das vom Steuerinterpreter angesteuert wird, samt Straßenbahnen nötig sind. Wie schon an der graphischen Anordnung zu erkennen ist, gliedert sich die Struktur des Pakets „sim“ in drei Bereiche:

- Gleisnetz-Elemente + Simulator
- Schnittstelle zu den Treibern
- Einlesen des Gleisnetzes

Nun folgt eine kurze Beschreibung der Klassen und Erklärung der Vererbungsstruktur.

- Gleisnetz-Elemente + Simulator
  Die Gleisnetz-Elemente beinhalten alle Gleiselemente aus realen Straßenbahnnetzen, die sich auf oder neben der Strecke befinden und elektronisch ansteuerbar oder abfragbar sind und für die sichere Fahrt eine Rolle spielen. Des Weiteren werden im Folgenden die Route und der Simulator definiert.

- Hardware
  „Hardware“ bildet die Vaterklasse für alle Elemente eines Gleisnetz, ausgenommen den Gleisen selbst (s. „TrackElement“). Abgeleitete Instanzen der abstrakten Klasse „Hardware“ besitzen eine eindeutige Identifikation und Ortskoordinaten.
- SwitchableHW
  „SwitchableHW“ ist eine abstrakte Klasse für schaltbare / steuerbare Hardware-Elemente. Diese Klasse enthält als Membervariablen Typ, Zustand, einen angeforderten Zustand und die Schaltzeit und erbt von der Klasse „Hardware“.
- Turnout
  „Turnout“ simuliert eine Weiche. Diese Klasse ist abgeleitet von „SwitchableHW“ und implementiert das Interface „Linkable“. Eine Weiche hat einen bestimmten Typ und kann bestimmte Zustände annehmen. Der angeforderte Zustand wird eingenommen, nachdem die Schaltzeit verstrichen ist. Es gibt eine Lock-Variable, welche die Weiche für automatisches Umstellen sperrt (Weichensperrkreis).
- Slipswitch
  „Slipswitch“ simuliert eine Kreuzungsweiche. Diese Klasse koppelt zwei bzw. vier Instanzen von „Turnout“ und eine von „Crossing“ zusammen und sorgt dafür, dass diese als ein gemeinsames Element bezüglich des Umschaltens und der Kollisionsabfrage behandelt werden.
- Signal
  „Signal“ simuliert ein Lichtsignal, das von einem Straßenbahnfahrer gesehen wird. Die Klasse erbt von „SwitchableHW“ und „Linkable“ und hat ähnlich wie „Turnout“ einen Typ, annehmbare Zustände, einen angeforderten Zustand und eine Schaltzeit.
- Mark
  „Mark“ bildet einen Stützpunkt im Gleisnetz, an dem die Gleisstücke verknüpft sein können. Die Klasse erbt direkt von „Hardware“ und „Linkable“ und hat keine weiteren Eigenschaften.
- Crossing
  „Crossing“ stellt eine Kreuzung im Gleisnetz dar, bei der die sich schneidenden Gleise auf gleicher Höhe sind. Diese Klasse erbt direkt von „Hardware“ und „Linkable“ und hat keine weiteren Eigenschaften.
- Sensor
  „Sensor“ erbt direkt von „Hardware“ und bildet eine abstrakte Vaterklasse für alle Sensortypen. Sensoren können aktiviert und deaktiviert werden.

- RouteRequestSensor
  „RouteRequestSensor“ erbt direkt von „Sensor“. Dieser Typ Sensor löst aus, wenn eine Straßenbahn bei der Überfahrt ein Routentelegramm schickt.
- StateSensor
  „StateSensor“ erbt direkt von „Sensor“ und erweitert diesen um die Eigenschaft, dass es möglich ist, festzustellen, ob sich eine Tram auf dem Sensor befindet.
- StateDirectionSensor
  „StateDirectionSensor“ erbt und erweitert somit den „StateSensor“. Dieser Sensor kann zusätzlich die Richtung feststellen, in die sich die Tram bewegt oder in die sich die letzte Tram bewegt hat.
- ToggleSensor
  „ToggleSensor“ erbt direkt von „Sensor“. Bei Überfahrt dieses Sensors schaltet ein ToggleBit um.
- ToggleDirectionSensor
  „ToggleDirectionSensor“ erweitert „ToggleSensor“ um die Möglichkeit der Richtungserkennung (vgl. „StateDirectionSensor“).
- Linkable
  „Linkable“ stellt eine gemeinsame Schnittstelle für die Klassen dar, die im Gleisnetz die Gleisstücke (TrackElements) miteinander verbinden (linken).
- Sourceable
  „Sourceable“ ist ein Interface, das von den Hardwareklassen implementiert wird, die potentielle Einfahrten in das Gleisnetz darstellen. Dies sind Mark und Sensor.
- Simulator
  „Simulator“ bildet die Einheit, die die anderen Klassen steuert, überwacht und abfragt. Er kennt alle anderen Klassen, startet den Steuerinterpreter und bietet zentrale Methoden an, die aufgerufen werden, wenn in der Simulation ein Fehler auftritt.
- TrackElement
  „TrackElement“ simuliert ein Gleisstück, was sich zwischen zwei Hardware--Elementen befindet. So ein Gleisstück hat eine bestimmte Länge und kennt die beiden Elemente, die an den Enden liegen.
- Tram
  „Tram“ simuliert eine Straßenbahn, die aus Kopf und Ende besteht. Sie fährt eine „Route“ entlang, reagiert auf Signale und kann mit einer anderen Straßenbahn zusammen stoßen.

  - Route
    „Route“ enthält die Informationen, über welche Sensoren eine Route führt und wie welche Weichen geschaltet sein müssen, damit die Route richtig abgefahren wird.
  - NetworkEndException
    „NetworkEndException“ ist von „Exception“ abgeleitet und ist lediglich vorhanden, um einen speziellen Exception-Typ zu haben.

- Schnittstelle zu den Treibern
  Die Schnittstelle zu den Treibern besteht aus zwei Klassen, die für das Empfangen und Senden von Daten zuständig sind.

  - DriverRequestServer
    „DriverRequestServer“ ist verantwortlich für das Empfangen von Anforderungen, die der Steuerinterpreter an die Treiber geschickt hat und die von diesen in Strings übersetzt worden sind.
  - StatusSender
    „StatusSender“ sendet den aktuellen Status eines Elements als Nachricht an die Treiber, die diese Information wiedrum in das Shared Memory eintragen.

- Einlesen des Gleisnetzes
  Da es möglich sein soll, den Simulator auf verschiedenen Gleisnetzen laufen zu lassen (s. S. 334), ist es notwendig, ein Gleisnetz automatisch einlesen und die Datenstruktur des Simulators daraus erstellen zu können.

  - Lexer
    „Lexer“ liest die Gleisnetz-Datei ein und fasst einzelne Zeichen zu vordefinierten Token zusammen. Der Klasse „Lexer“ wird automatisch generiert. Als Spezifikationsdatei für diese Klasse wird die Datei „lexer.def“ zu Grunde gelegt.
  - Parser
    „Parser“ bekommt vom Lexer das Gleisnetz in Token-Form und arbeitet dieses anhand einer Grammatik durch. Relevante Informationen gibt er an den „BestTramNetworkBuilderOfTheUNiverse“ weiter. Auch die Klasse „Parser“ wird automatisch erstellt und hat als Spezifikation die Datei „parser.def“.
  - Sym
    „Sym“ enthält alle Terminale, die Lexer und Parser verwenden. Auch diese Klasse wird automatisch generiert auf Grundlage der „parser.def“.
  - BestTramNetworkBuilderOfTheUNiverse
    „BestTramNetworkBuilderOfTheUNiverse“ erstellt die Datenstruktur im Si-

mulator, d. h. er baut die Klassenstruktur aufbauend auf dem geparsten Gleisnetz auf, so dass das statische System des Simulators erzeugt wird.

7.8.4.2.2 Paket „gui“ Im Paket „gui“ sind sämtliche Elemente vorzufinden, die mit der graphischen Darstellung des Simulators zu tun haben. Diese sind:

- MainFrame
  „MainFrame“ realisiert das Hauptfenster der Darstellung, in dem auch das Bedienungsmenü vorliegt.
- ImageComponent
  „ImageComponent“ ist für die Visualisierung der Simulation zuständig, hier befinden sich die Methodem zum Zeichnen der Elemente.
- NewSimDialog
  „NewSimDialog“ ist einer von mehreren Konfigurationsdialogen zum Starten einer Simulation.
- SensorSettingDialog
  „SensorSettingDialog“ ist einer von mehreren Konfigurationsdialogen zum Starten einer Simulation.
- SignalSettingDialog
  „SignalSettingDialog“ ist einer von mehreren Konfigurationsdialogen zum Starten einer Simulation.
- TramCountDialog
  „TramCountDialog“ ist einer von mehreren Konfigurationsdialogen zum Starten einer Simulation.
- TramSettingDialog
  „TramSettingDialog“ ist einer von mehreren Konfigurationsdialogen zum Starten einer Simulation.
- TurnoutSettingDialog
  „TurnoutSettingDialog“ ist einer von mehreren Konfigurationsdialogen zum Starten einer Simulation.
- AboutDialog
  „AboutDialog“ ist ein allgemeiner Anzeigedialog zur Ausgabe von Informationen.

7.8.4.3 Flexibilität – Verarbeitung verschiedenster Gleisnetze

Damit der Simulator seiner Aufgabe gerecht werden kann, muss er verschiedenste Gleisnetze simulieren können. Um sicherzustellen, dass er das gleiche Gleisnetz als Grundlage hat, wie der mit ihm verbundene Steuerinterpreter, muss er inhaltlich gleiche Gleisnetz-Konfigurationsdaten bekommen. Diese liegen in einer Datei im Format TND vor. Der Inhalt dieser TND-Dateien befolgt eine festgelegte Grammatik, die wiederum in EBNF-Form im Projekt TRACS aufgeschrieben wurde (siehe A.1 auf Seite 453 und A.2 auf Seite 456). Für den SI wird diese Datei mittels eines Compilers in Binärdaten übersetzt. Der Simulator hingegen wird diese Datei direkt einlesen und somit auf den gleichen Daten arbeiten.
Das erste nun folgende Unterkapitel beschreibt den Ablauf des Einlesens einer Datei. Das anschließende Kapitel beschreibt die Gründe, warum wir uns für die Benutzung von JFlex und CUP entschieden und keinen eigenen Compiler programmiert haben.

7.8.4.3.1 Einlesevorgang einer TND-Datei Das Einlesen einer TND-Datei mit Hilfe von Lexer und Parser ist ein Ablauf der in mehrere Schritte unterteilt ist. Die Eingabe besteht aus einer TND-Datei und das Ergebnis soll eine instanziierte Klassenstruktur in Java™sein. Auf dem Weg zu diesem Ziel werden folgende Schritte durchlaufen, welche zur Veranschaulichung in Abb. 7.63 auf der nächsten Seite dargestellt sind:

1. Lexikalische Analyse
   Der Scanner (oder Lexer) liest die Gleisnetzbeschreibung, die in Form einer TND in einer Datei vorliegt, ein und gibt beim Abruf des Parsers ein Symbol nach dem anderen an diesen weiter (vgl. [ASU99, Abb. 3.1 auf S. 102]).

2. Syntaktische Analyse
   Der Parser verarbeitet die Symbole des Scanners. Er erkennt fehlerhafte Eingaben und beschreibt diese Fehler möglichst präzise. Bei dieser Analyse entscheidet unser Parser über die Relevanz der einzelnen Informationen. Nur die für den Simulator und die für die Prüfung des CAD-TND-Konverters entscheidenden Informationen aus der TND werden anschließend an den BestTramNetworkBuilderOfTheUNIverse weitergegeben.

3. Erzeugung der Klassenstruktur
   Der BestTramNetworkBuilderOfTheUNIverse bekommt die relevanten Informationen vom Parser und erzeugt (instanziiert) aus ihnen die Klassenstruktur, auf der der Simulator letztendlich arbeiten kann.

7.8.4.3.2 Pro und Kontra Compiler Konstruktions Werkzeuge Damit die Anforderungen aus Kapitel 7.8.2.1 auf Seite 314 erfüllt werden können, muss eine Text-

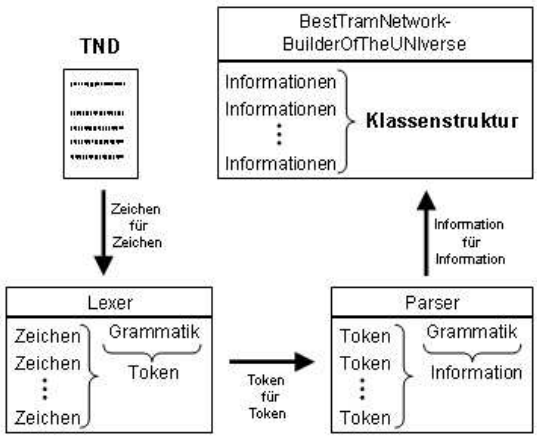


Abbildung 7.63: Ablauf: TND einlesen und verarbeiten

datei, deren Form in der TND festgehalten ist, eingelesen werden können. Dieser Einlesevorgang ist kein trivialer Vorgang, denn die Grammatik, in der die Datei geschrieben ist, ist kompliziert und damit auch nicht leicht zu überprüfen. Des Weiteren müssen Information zu ein und demselben Bauteil von verschiedenen Stellen parallel eingelesen und verarbeitet werden.
Dieser Einlesevorgang benötigt demzufolge einen eigenen Compiler (bzw. Übersetzter) oder einen – mit Hilfe von Compiler Konstruktions Werkzeugen (Lexer- und Parser-Generatoren) – generierten Compiler. Dieses Kapitel beantwortet die Frage, warum die Arbeitsgruppe sich *für* die Benutzung von Compiler-Konstruktions-Werkzeugen (CKW) entschieden hat. Die Vor- und Nachteile der Benutzung dieser Werkzeuge sind in Tabelle 7.1 auf der nächsten Seite grob zusammengefasst und werden gegenübergestellt.
Das Mehr an Programmieraufwand ohne CKWs ist offensichtlich, da CKWs ja Quellcode generieren und somit einen Teil des Programmieraufwands ersetzen. Gleichfalls ersichtlich ist der erhöhte Einarbeitungsaufwand bei der Verwendung von CKWs, da man sich hier erst einmal mit der Beschaffung, der Installation, der Verwendung und mit der Spezifikation beschäftigen muss, die ein CKW braucht, um daraus etwas zu generieren.
Bei einem Verzicht auf CKWs ist die Vermischung von Scanner und Parser möglich. Zwar ergeben sich dadurch teilweise Optimierungsmöglichkeiten durch sehr individuelle Anpassung der Implementierung, gleichzeitig erhöht sich aber auch die Gefahr, dass die Vermischung zweier, in der Theorie eigenständige, Teile die Fehlerwahrscheinlichkeit in der Implementierung erhöht. CKWs trennen hier im Allgemeinen ganz klar (auch um ihre eigene Komplexität in Grenzen zu halten) und diese strukturierte Generierung verhindert logische Fehler im Aufbau des Compilers (vorausgesetzt die dem Generator zugrunde liegende Grammatik ist korrekt).
Eine eigene Implementierung des Compilers ermöglicht kein direktes Herauslesen der Grammatik aus dem Quellcode, während bei der Benutzung von CKWs solch eine Gram-

Compiler Konstruktions Werkzeuge	
ohne	**mit**
Unterschiede	
mehr Programmieraufwand – Scanner und Parser müssen komplett von Hand geschrieben werden	mehr Einarbeitungsaufwand – Einarbeitung in die Spezifikationsdateien und in die Benutzung der Generatoren
(theoretisch) Optimierungsmöglichkeiten durch individuellere Gestaltung	geringere Fehleranfälligkeit durch strukturierte Generierung
eingelese Grammatik nur schwer zu erkennen	eingelesene Grammatik anhand der Spezifikation klar ersichtlich
bei Änderungen an der einzulesenden Grammatik muss:	
die Implementierung des Parsers überarbeitet werden	nur die Spezifikation angepasst werden
	keine Neuerfindung von bereits erforschten Algorithmen
nur LL(1)-Compiler mit angemessenem Aufwand möglich	LALR(1)-Compiler möglich
Gemeinsamkeiten	
Code Generierung muss manuell implementiert werden	

Tabelle 7.1: Vergleich: pro und kontra Compiler Konstruktions Werkzeuge

matik / Spezifikation als Grundlage für die Generierung benutzt wird und folglich klar erkennbar ist. Bei Änderungen der zugrunde liegenden Grammatik ist es bei Benutzung von CKWs folglich ersichtlich, ob die Änderungen bereits eingefügt worden sind oder nicht. Bei solchen Änderungen muss bei einem selbst implementierten Compiler die Implementierung angepasst werden, während bei einem generierten Compiler lediglich die Grammatik geändert werden und der Generierungsschritt neu ausgeführt werden muss. Ob mit oder ohne Compiler-Generator, der Teil, in dem aus der erkannten Grammatik Code generiert wird, muss immer manuell geschrieben und angepasst werden. Dieser Aspekt dient folglich nicht der Entscheidungsfindung.
Zusammengefasst lässt sich sagen, dass eine Generierung eines Compilers zwar die Komplexität des Gesamtprogramms erhöht, dies aber zugunsten der Struktur und der verminderten Fehleranfälligkeit geschieht. Seit der Entwicklung der ersten Compiler in den fünfziger Jahren „wurden systematische Techniken entdeckt, um mit vielen der wichtigen Aufgaben fertig zu werden, die sich während einer Compilierung ergeben“ [ASU99, S. 2]. Viele Techniken wurden seitdem optimiert und es ist fraglich, ob es Sinn macht, wenn man diese Techniken ‚neu erfindet‘ und einen Compiler komplett selbst schreibt.

7.8.4.3.3 Projektinterne Entscheidungsgründe Außer dem allgemeinen Vergleich der Ansätze müssen auch projektinterne Entscheidungsgründe berücksichtigt werden, da es bereits eine Teilgruppe *Compiler* gibt und diese sich entschlossen hat, Lexer- und Parser-Generatoren zu verwenden, nämlich *flex* als Lexer- und *bison* als Parser-Generator. Da diese Tools für C und nicht für Java™entwickelt wurden, kann die Arbeitsgruppe sie nicht direkt benutzen. Aber wenn *JFlex* und *CUP* (Java™Based Constructor of Useful Parsers) verwendet werden, die sehr stark an ihre C-Pendants angelehnt sind, dann lässt sich sicherlich Know-how der Compiler-Gruppe und auch ein Großteil der Spezifikationen (in denen z. B. die einzulesende Grammatik steht) verwenden.
Des Weiteren muss im Projekt berücksichtigt werden, dass es jederzeit dazu kommen kann, dass Projektteilnehmer das Projekt verlassen. In diesem Fall muss gewährleistet sein, dass keine nicht schließbare Lücke entsteht bzw. dass kein Quellcode von niemandem mehr verstanden wird. Ein vollständig selbst geschriebener Compiler, der eventuell von einer einzigen Person geschrieben wurde, lässt sich nur bei sehr strukturiertem und gut dokumentiertem Quellcode gut anpassen. Da die Generierung sehr strukturiert vorgeht und die Compiler Konstruktions Werkzeuge bereits ausführlich getestet und dokumentiert sind, sollte die Einarbeitung bei einem – zumindest zum Teil – generierten Compiler einfacher sein.

7.8.4.3.4 Fazit Wir haben uns für die Benutzung von Compiler-Konstruktions-Werkzeugen entschieden, um Fehler bei einer eigenen Implementierung zu vermeiden und um schneller und einfacher auf Änderungen der einzulesenden Grammatik reagie-

ren zu können. Der Simulator soll ein Test für den Steuerinterpreter sein und wenn der Simulator auf fehleranfälligen Konzepten basieren würde, dann müsste bei vielen Fehlern eventuell erst einmal nachgeschaut werden, ob der Fehler nicht beim Simulator liegt – aus diesem Grund vertrauen wir lieber auf die langjährige Weiterentwicklung der Theorie über Compilerbau und verwenden die genannten Tools, um auf Grundlage dieser Theorie basierend unseren Compiler zu erstellen.

7.8.4.4 Simulation

Den Kern des Simulators bildet die Simulation (s. Anforderungen 7.8.2.2 auf Seite 316). Der rundenbasierte Ablauf wird im Absatz *Ablauf einer Runde* erklärt. Der Simulator soll nicht nur Weichen und Signale so stellen können, wie der Steuerinterpreter sie anfordert (*Annahme von ANforderungen zur Stellungsänderung*), sondern Weichen, Sensoren und Signal liefern auch Änderungen ihres Zustands an die Treiber, bzw. den Steuerinterpreter zurück (*Zustandsänderungen zurückmelden*). Damit Sensoren auslösen können, müssen Straßenbahnen auf dem Gleisnetz bewegt werden. Wie sich eine Straßenbahn fortbewegt und wie die darunter liegende Datenstruktur aussieht, wird im Absatz *Bewegungsalgorithmus* näher erläutert.
Außerdem muss die Simulation initialisiert werden, insbesondere müssen die Straßenbahnen gesetzt und ihnen eine Fahrtroute zugewiesen werden, damit die Simulation korrekt ablaufen kann. Laut Anforderungen soll es zusätzlich möglich sein, ein fehlerhaft agierendes Gleisnetz simulieren zu können (s. Anforderungen 7.8.2.2 auf Seite 316). Das Konzept der Initialisierung und der fehlerhaften Aktion ist in den Absätzen *Initialisierung einer Simulation* und *Simulation von fehlerhaft agierendem Gleisnetz* zu finden.

7.8.4.4.1 Ablauf einer Runde Der Ablauf einer Runde im Simulator gliedert sich in mehrere Bereiche:

- Anforderungen holen
- ggf. Schäden „verursachen“
- stellbare Hardware-Elemente aktualisieren
- Straßenbahnen aktualisieren
- Veränderungen der Hardware-Elemente senden
- Output beschädigter Sensoren generieren
- Neue Bahnen starten
- Zeitstempel holen und warten

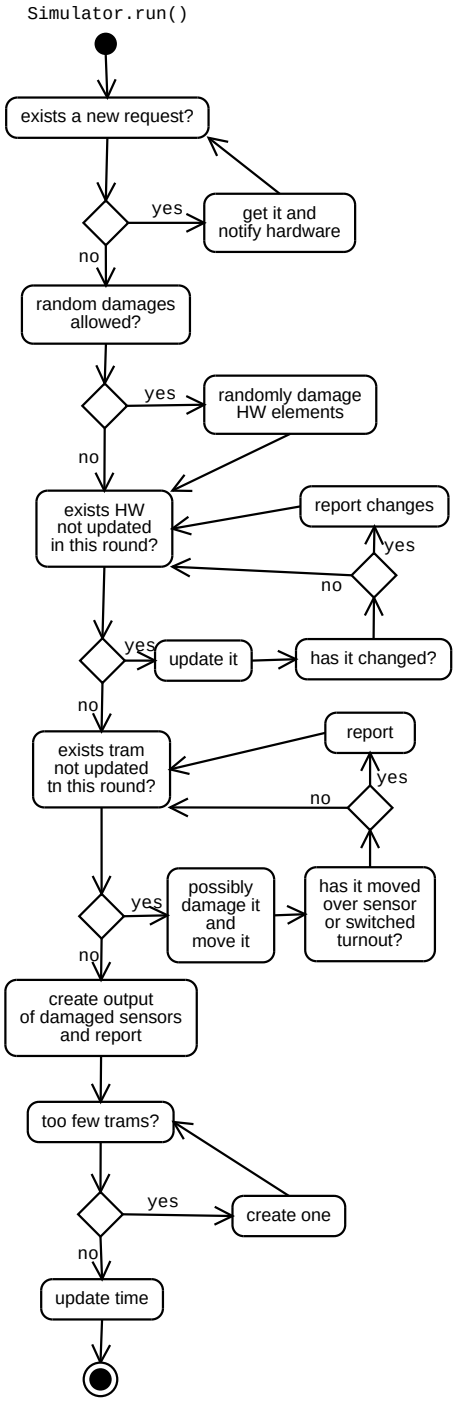


Abbildung 7.64: Aktivitätsdiagramm: Simulator.run()

Der Simulator bildet das Bindeglied zwischen der Schnittstelle zu den Hardware-Treibern und den einzelnen Hardware-Elementen in der Simulation. Es soll keine direkte Kommunikation zwischen der Treiber-Schnittstelle und z. B. einer Weiche erfolgen, damit der Ablauf übersichtlich, nachvollziehbar und leichter überprüfbar ist.

1. Anforderungen holen
   Zu Beginn einer Runde prüft der Simulator, ob neue Anforderungen des Steuerinterpreters an die Hardware gesendet wurden. Falls Ja, so werden die Anforderungen einzeln vom Empfangsserver abgeholt und ausgeführt, d.h. die Anforderung wird an das entsprechende Hardware-Element weitergeleitet.
   Ein beispielhafter Ablauf könnte wie folgt aussehen: Nehmen wir an, der Steuerinterpreter ‚entscheidet' sich zur Umschaltung einer Weiche und beschreibt den entsprechenden Speicherbereich im Shared Memory. Diese Änderung wird vom Treiber erkannt, der daraufhin einen String mit der Anforderung zusammensetzt. Dieser String wird dann an den DriverRequestServer gesendet, welcher im Simulator die ‚Anforderungsannahme' darstellt. Der String wird also eingelesen, auf Korrektheit geprüft und im Falle der Korrektheit in einer Warteliste abgelegt.
   Wie bereits beschrieben, prüft der Simulator in jedem Rundendurchlauf, ob neue Anforderungen anliegen – ist das der Fall, so holt er sich diese aus der Warteliste, extrahiert die benötigten Informationen aus dem String und ruft auf dem Hardware-Element, für das die Anforderung gilt (in unserem Fall also eine Weiche), eine Methode auf, die diesem Element die Anforderung übergibt. Zusätzlich wird der Zeitpunkt der Anforderung gespeichert, damit das Element später nach der richtigen Reaktionszeit umschalten kann. (siehe 7.65).

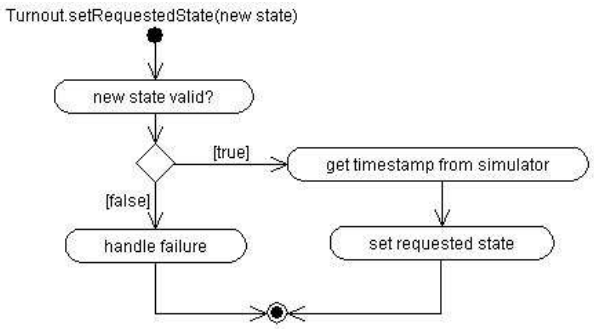


Abbildung 7.65: Aktivitätsdiagramm: Turnout.setRequestedState()

2. Hardware beschädigen
   Ist es gewünscht, dass Hardware-Elemente ausfallen können, so werden nach einem Zufallsverfahren Elemente „beschädigt". Solchermaßen gekennzeichnete Elemente können bei der Aktualisierung im nächsten Schritt anderes als das Normalverhalten zeigen.

3. stellbare Hardware-Elemente aktualisieren
   Anschließend werden alle stellbaren Hardware-Elemente aktualisiert. Bei einer Aktualisierung kann eine stellbare Hardware ihren Zustand ändern, wenn genügend Zeit seit der Annahme einer Anforderung für dieses Element vergangen ist (in der Regel werden die Elemente nach der Hälfte ihrer in der TND festgehaltenen maximalen Schaltzeit umgestellt). Beschädigte Elemente ändern ihren Zustand nicht, auch wenn das erwartet werden sollte, oder sie ändern ihn spontan und unerwartet. Der Simulator merkt sich alle veränderten Hardware-Elemente, da er die Veränderungen dem Steuerinterpreter in einem späteren Schritt mitteilen wird.

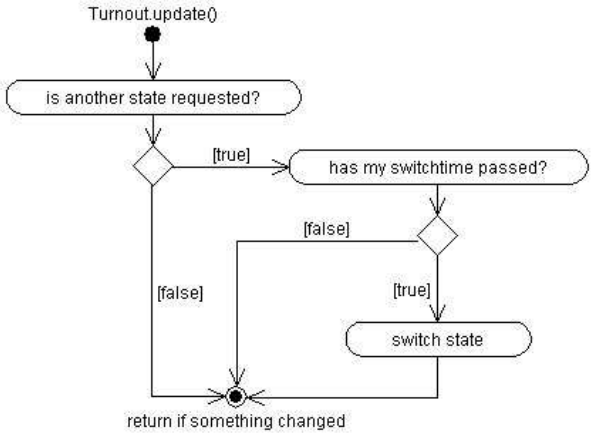


Abbildung 7.66: Aktivitätsdiagramm: Turnout.update()

In jedem Durchlauf des Simulators wird auf jedem Hardware-Element ein Update ausgeführt, d.h. es wird geprüft, ob in der Vergangenheit eine Anforderung eingegangen ist, die noch nicht umgesetzt wurde, und ob der Zeitpunkt zur Umsetzung gekommen ist (in der Regel wird nach der Hälfte der in der TND eingetragenen maximalen Schaltzeit umgeschaltet). Falls dem so ist, wird die Umschaltung vorgenommen und gleichzeitig vermerkt, dass eine solche stattfand.
Weiterhin wird beim Hardware-Update im Fall von Weichen und Kreuzungen ein evtl. „Besetzt“-Zustand zurückgesetzt, falls im vorigen Zyklus die sich auf dem Element befindliche Bahn von diesem entfernt hat. Dieses Rücksetzen kann nicht sofort im Zeitpunkt des Verlassens ausgeführt werden, da ggf. im selben Zyklus andere Bahnen auf das Element auffahren würden. Die Reihenfolge, in der in dem Zyklus die Bahnen bewegt würden, hätte dann Auswirkungen darauf, ob eine Kollision erkannt wird oder nicht. Mit diesem nachgelagerten Rücksetzen wird erreicht, dass in der gegebenen Situation in jedem Fall eine Kollision erkannt wird.

In der Grafik 7.66 ist der Normalfall für ein unbeschädigtes Element zu sehen. Ein

defektes Element wird dagegen nicht umschalten oder dann umschalten, wenn es nicht verlangt ist.

4. Straßenbahnen aktualisieren
   Nach der stellbaren Hardware werden als nächstes die Straßenbahnen fortbewegt. Durch die Straßenbahnen können sich auch die Zustände von Hardware-Elementen verändern ((De-)aktivieren eines Sensors, passives Überfahren einer Weiche). Der Simulator sammelt die durch die Straßenbahnen veränderten Hardware-Elementen und fügt sie der Liste, die er im vorherigen Schritt erstellt hat, hinzu. Auch Straßenbahnen werden ggf. beschädigt und bleiben dann entweder stehen oder beachten Signale nicht mehr. Der Bewegungsalgorithmus wird in der Folge genauer beschrieben.

5. Veränderungen der Hardware-Elemente senden
   Wenn sich etwas im Gleisnetz verändert hat, so sendet der Simulator nun die Veränderungen an die Hardware-Treiber. Die Zustandsänderungen werden zurückgemeldet, indem der Simulator von allen geschalteten / ausgelösten Hardware-Elementen (die er sich ja beim Umschalten gemerkt hat) den neuen Zustand aufnimmt und jeweils einen String gemäß dem dafür definierten Format zusammensetzt. Diese Strings werden in eine Sendewarteliste eingetragen.
   Ein eigenständiger Thread, der StatusSender, prüft ständig den Inhalt dieser Liste – findet er etwas vor, wird dieser String an den Treiber geschickt, der seinerseits den Inhalt des Shared Memory entsprechend abändert, so dass der Steuerinterpreter die Schaltung erkennen kann. Ein eigener Thread wurde deshalb gewählt, um im Fall von Störungen beim Senden den Simulator dennoch weiter laufen lassen zu können.

6. Output beschädigter Sensoren generieren
   Da Sensoren nicht zu den stellbaren Elementen zählen, werden sie im obigen Schritt nicht aktualisiert. Nichtsdestotrotz können auch sie beschädigt sein, und Meldungen abgeben, wenn eigentlich keine erfolgen sollten. Das geschieht in diesem Schritt.

7. Neue Bahnen starten
   Falls zu wenig Bahnen im Netz sind, werden an dieser Stelle neue Bahnen eingesetzt. Bei Bahnen mit vorbestimmten Routen wird eine von diesen gestartet, im anderen Fall wird eine Route zufällig gewählt.

8. Zeitstempel holen und warten
   Von der Systemuhr wird die aktuelle Zeit geholt und auf das Ende des Simulatorschritts gewartet. Das Warten kann unterschiedlich lange ausfallen, je nachdem wieviel Zeit die Berechnungen dieser Runde gekostet haben. Da im Endeffekt aber

jede Runde gleich lange dauern soll, wird das mit einem variablen Warteanteil ausgeglichen. (Die Rundenzeit ist so gewählt, dass genügend Luft besteht.)

7.8.4.4.2 Bewegungsalgorithmus Zwei wichtige Voraussetzungen zur Simulation von (Straßen-) Bahnen sind, dass man bestimmen kann, an welchem Ort sie sich gerade befinden und dass man die Bewegung simulieren kann. Da wir uns für eine rundenbasierte Programmablaufstruktur entschieden haben (s. S. 368), müssen wir die Bahnen in konstanten Schritten bewegen. Nun folgen Beschreibungen über

- die Programmstruktur
- und den Algorithmus,

die wir für die Lösung der Simulationsanforderungen verwenden. Bei der Entwicklung dieses Konzeptes legen wir eine hohe Priorität auf die zeitliche Optimierung der Berechnungsschritte, die während der Simulation stattfinden – bewusst vernachlässigen wir dabei die Zeit für die Berechnungen, die vor Beginn der Simulation durchgeführt werden müssen und den allgemeinen Speicherverbrauch.

- Programmstruktur

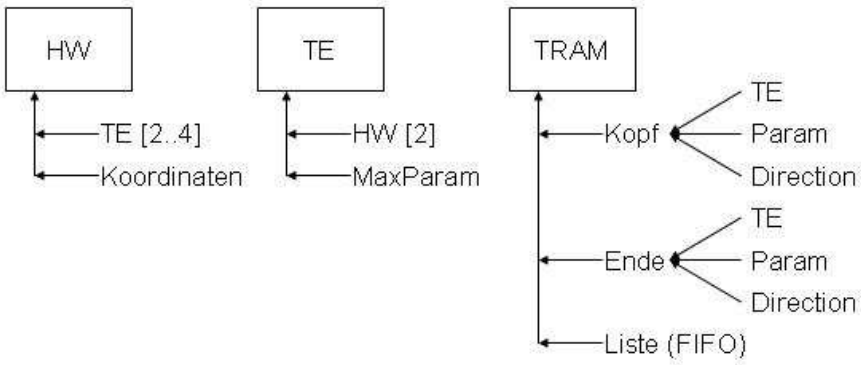


Abbildung 7.67: Programmstruktur für Bewegungsalgorithmen

Die Programmstruktur ist dargestellt in Abbildung 7.67. Die Instanzen der Klasse `HW` (Hardware) bilden die Stützpunkte des Gleisnetzes – zwischen ihnen existieren gerade Verbindungen (TrackElement, TE). Damit lassen sich zwar keine richtigen Kurven modellieren, doch diese Ungenauigkeit spielt für den Steuerinterpreter und damit auch für den Simulator keine Rolle. Nur Signale sind nicht durch TEs mit den anderen HWs verbunden, da keine Route direkt durch ein Signal hindurchgeht. Zwar müssen sie für eine Interaktion mit der Bahn mit dem Gleisnetz verbunden sein – die Bahn muss wissen wo sie halten muss – aber das wird nicht für die

Bewegung einer Bahn an sich benötigt und wird deshalb in einem späteren Kapitel behandelt. Ein HW beinhaltet folgende Informationen:

- TE[2..4]
  Dieses Array von TE beinhaltet die angrenzenden Geradenstücke, deren Anfang / Ende an dieses HW gebunden sind. Wenn die Größe des Arrays gleich 2 ist, dann ist dieses HW ein Verbindungsstück auf einer einzelnen Strecke. Wenn die Größe gleich 3 ist, dann handelt es sich bei diesem HW um eine Weiche und beim Index 0 befindet sich das TE auf der *stem*-Seite und bei den anderen Indizes die TEs der *branch*-Seite der Weiche.
  Bei der maximalen Größe von 4 handelt es sich bei diesem HW um eine Kreuzung oder eine Doppelweiche. Bei einer Kreuzung muss darauf geachtet werden, dass an den Indizes 0 und 1 zwei TEs stehen, die direkt ineinander übergehen – für die Indizes 2 und 3 gilt das gleiche. Im Gegensatz dazu muss bei einer Weiche darauf geachtet werden, dass sich beim Index 0 das TE der *stem*-Seite befindet und bei den anderen Indizes die TEs der *branch*-Seite befinden.
- Koordinaten
  Die Koordinaten der einzelnen HWs werden benötigt als Positionsangabe – ohne diese wäre eine Visualisierung gar nicht möglich. Gleichfalls werden sie dazu gebraucht, um z. B. die Länge eines TEs aus zurechnen.

Die Instanzen der Klasse `TE` (TrackElement) enthalten die Informationen für die Geradenstücke zwischen zwei HWs. TEs verbinden jeweils zwei HWs miteinander. Ein TE selbst ist kein aktives Element des Gleisnetzes – es wird passiv für die Verbindung der HWs und für die Positionsbestimmung der Bahnen benötigt und benutzt. Ein TE enthält folgende Informationen:

- HW[2]
  Ein TrackElement ist ein Geradenstück zwischen genau zwei HWs, die an dieser Stelle festgehalten werden. Die genaue Positionsbestimmung auf einem TE funktioniert mit Hilfe von Parametern, die im Folgenden genauer beschrieben werden. Wichtig an dieser Stelle anzumerken ist, dass beim HW am Index 0 der Parameter gleich 0 ist und beim HW am Index 1 der Parameter gleich *MaxParam* ist. Anhand dieser Festlegung wird die Orientierung eines TEs bestimmt.
- MaxParam
  Parameter werden zur genauen Positionsbestimmung auf einem TE benötigt. Es handelt sich um Ganzzahlwerte zwischen 0 und MaxParam. Zwischen den Parametern i und $i + 1$ ist immer der gleiche Abstand – dieser ist für alle TEs der gleiche und wird zentral festgehalten. Eine Fortbewegung der Bahn,

ausgedrückt in der Differenz der Parameter, ist mindestens 1 pro Runde, eine kleinere Einheit gibt es nicht. Aus diesem Grund muss die Einheit so klein gewählt werden, dass die Simulation eine flüssige Bewegung darstellt. Die Größe dieser Einheit wirkt sich weder auf die benötigte Speichergröße des Simulators, noch auf die Rechenzeit aus und kann insofern fast beliebig gewählt werden.

Die dritte, für die Bewegung der Bahnen notwendige, Klasse heißt `Tram` (Straßenbahn). Sie stellt ein aktives Element im Simulator dar. Sie entscheidet (nach eingestelltem Setup) selbstständig, ob sie fährt oder nicht. Für die Fortbewegung an sich benötigt eine Bahn folgende Informationen:

- Kopf
  Der *Kopf* beschreibt die Spitze der Tram. Zwischen ihm und dem *Ende* befindet sich die gesamte Tram. Eine Tram bewegt sich immer in Richtung des Kopfes.
  - TE
    Das TrackElement, auf dem sich der Kopf der Tram befindet.
  - Param
    Der Parameter, der angibt, wo auf dem TE sich der Kopf der Tram befindet.
  - Direction
    Die Richtung, in die sich der Kopf der Tram (und damit die gesamte Straßenbahn) auf dem aktuellen TE bewegt – also entweder in Richtung der kleiner werdenden Parameterwerte oder in die entgegengesetzte Richtung.
- Ende
  Zwischen dem *Kopf* und *Ende* der Tram befindet sich die Bahn. Die Informationen für das Ende der Tram werden getrennt festgehalten, da sich die Parameterwerte aufgrund einer Länge größer 0 der Tram unterscheiden. Da sich Kopf und Ende nicht zwingender maßen auf ein und demselben TE befinden (z. B. in dem Moment, wo der Kopf gerade ein neues TE erreicht hat) muss auch diese Information, sowie die Richtung getrennt festgehalten werden. Die Richtung muss auf jedem TE neu bestimmt werden, da Orientierung eines TEs zufällig, bzw. je nach Anordnung in der TND gewählt ist.
  - TE
    Das TrackElement, auf dem sich das Ende der Tram befindet.
  - Param
    Der Parameter, der angibt, wo auf dem TE sich das Ende der Tram befindet.

* Direction
  Die Richtung, in die sich das Ende der Tram auf dem aktuellen TE bewegt – also entweder in Richtung der kleiner werdenden Parameterwerte oder in die entgegengesetzte Richtung.

– Liste
  Diese Liste vom Typ *TE* enthält alle TrackElemente, auf denen sich – wenn auch nur ein Teil – die Straßenbahn befindet. Die Visualisierung wird hierdurch vereinfacht, falls ein TE existiert, dass vollständig von einer Tram besetzt ist und auf dem sich dennoch weder Kopf noch Ende der Tram befindet. Auf die Weise wird keine Art ‚Pathfinder' zwischen Kopf und Ende benötigt, der eventuell eine größere Menge von Rechenzeit in Anspruch nehmen würde. Die Länge der Liste ist damit auf ≥ 1 festgelegt.

- Algorithmen
  Im Folgenden werden die Algorithmen vorgestellt, die zur Bewegung der Trams notwendig sind. Zuerst wird die Methode „update()" vorgestellt. Ausgehend von dieser Funktion, werden dann die dort verwendeten Algorithmen („drive()", „checkStop()", „calculatePosition()", „driveInto()", „checkHeadCollision()", „checkCollision()" und „doFinish()") näher erläutert.

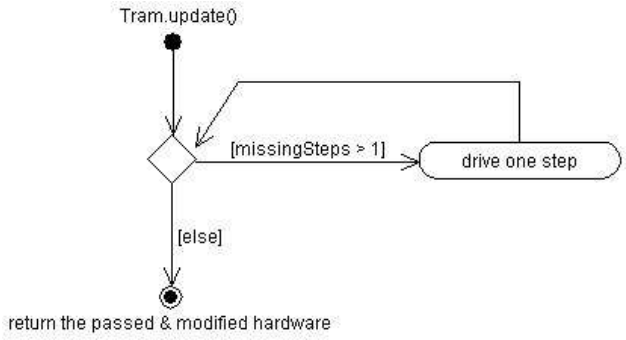


Abbildung 7.68: Aktivitätsdiagramm: Tram.update()

Tram.update() Der Simulator ruft bei jeder Straßenbahn die Methode „update()" auf, um sie fortzubewegen. Anhand der vor Start festgelegten Geschwindigkeit der Bahn wird berechnet, wie viele Schritte sie in diesem Simulatioszyklus fahren muss. Dies wird in der Variablen „missingSteps" abgespeichert. Dann prüft die Tram, ob sie sich fortbewegen darf. Falls Ja, ruft sie die Methode „drive()" auf. „drive()" wird so oft aufgerufen, bis die Straßenbahn keinen Schritt mehr

fortbewegt werden kann. Während der Fahrt werden die überfahrenen Hardware-Elemente, bei denen sich dadurch etwas geändert hat (Weichen und Sensoren), in der Tram-Instanz gespeichert, um später für den Simulator abrufbar zu sein.

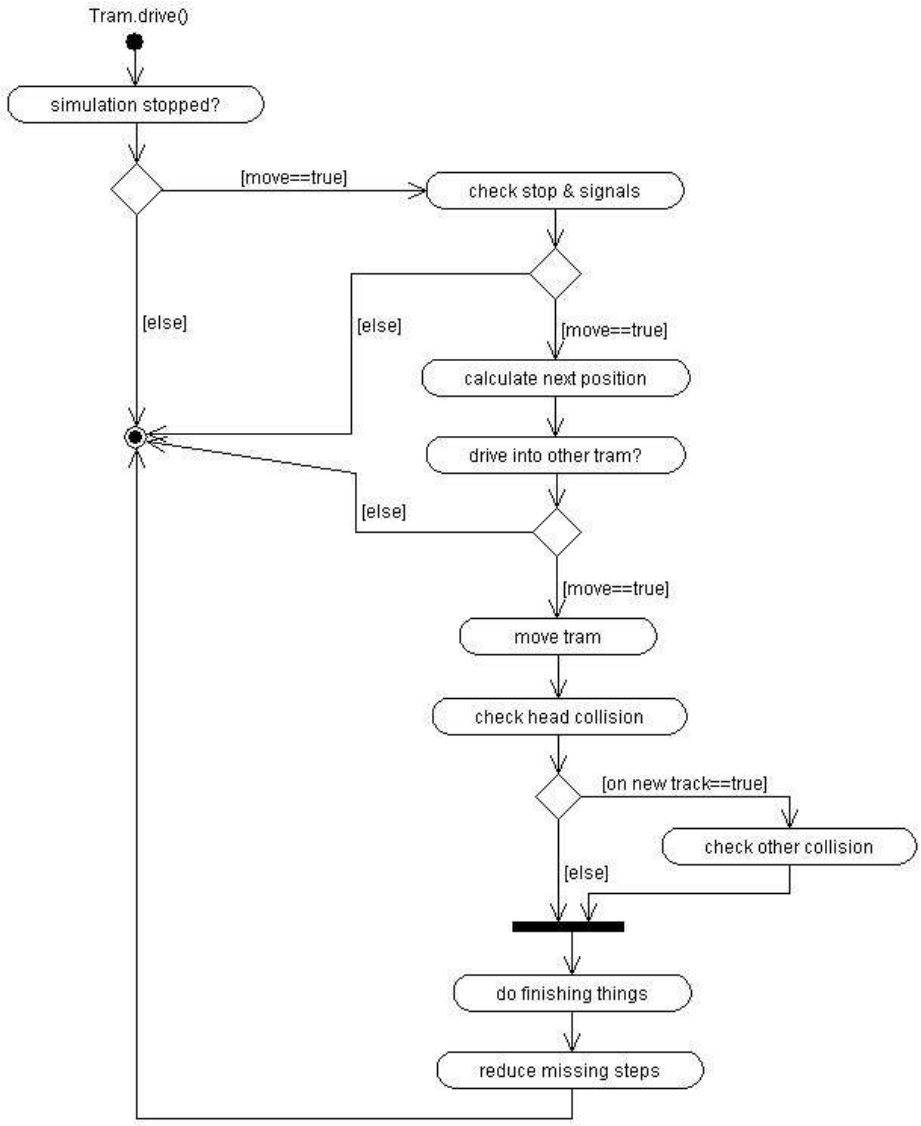


Abbildung 7.69: Aktivitätsdiagramm: Tram.drive()

Tram.drive() „drive()“ enthält den Ablauf, was passiert, wenn eine Straßenbahn um genau einen Schritt weiter fortbewegt werden soll. Falls die Bahn nicht im Modus „Signale nicht beachten„ ist, wird nach möglichen Haltesignalen geschaut und geprüft, ob die Tram aufgrund eines Signals halten muss (s. „checkStop()“).

Wenn die Straßenbahn halten muss, ist die Methode beendet. Wenn nicht, wird mit „calculatePosition“ die neue Position der Bahn errechnet. Dann wird geprüft, ob die Straßenbahn an der neu errechneten Position auf eine andere Tram auffahren würde („driveInto()“). Ist dies der Fall, so bricht die Methode ab. Ansonsten wird die Tram auf die neue Position bewegt. Dann erfolgt die Überprüfung, ob es aufgrund eines Fehlers vom Steuerinterpreter zu einem Zusammenstoß gekommen ist. Zuerst wird geprüft, ob zwei Straßenbahnen frontal zusammen gestoßen sind (s. „checkHeadCollision()“). Danach wird geprüft, ob die Tram ein neues Gleisstück betreten hat. In dem Fall kann es zu einer seitlichen Kollision mit einer anderen Bahn gekommen sein (s. „checkCollision()“). Nach den Kollisionsprüfungen behandelt „doFinish()“ die Reaktionen, die evtl. durch die Fahrt der Straßenbahn ausgelöst wurden. Die Anzahl der Schritte, die die Straßenbahn noch in dieser Runde fährt, wird um Eins erniedrigt.

Tram.checkStop() Diese Methode realisiert die Vorausschau eines Straßenbahnfahrers nach Signalen, vor denen er halten muss, und die Entscheidung, ob er direkt vor einem Haltesignal steht und halten muss. Der Algorithmus ist so aufgebaut, dass nur bis zum nächsten Hardware-Element geschaut wird. Da es aber auch sein kann, dass der Fahrer auf mehr als ein Signal in seiner Sichtweite achten muss, ruft der Algorithmus sich selbst iterativ auf, bis er am Ende der Sichtweite angekommen ist. Als Parameter werden dieser Methode die Position der Bahn und die Länge der Vorausschau übergeben. Ein Durchlauf von „checkStop()“ gliedert sich in drei Bereiche: die Registration von Stop-Signalen, der iterative Aufruf und die Entscheidung, ob die Straßenbahn in diesem Augenblick anhalten muss.

Zunächst holt sich die Methoden das Hardware-Element, auf das die Tram als nächstes fahren wird. Dann wird geprüft, ob dieser Hardware ein Signal zugeordnet ist. Falls Ja, wird der Zustand des Signals abgefragt. Wenn das Signal etwas anderes als Stop zeigt oder es gar kein Signal gibt, so wird als nächstes geprüft, ob an dieser Stelle zuvor ein Signal gesehen wurde. Falls Ja, dann wird jetzt vermerkt, dass dort kein Stop-Signal mehr ist. Gibt es ein Signal und zeigt es Stop an, so wird geprüft, ob die Straßenbahn noch weit genug entfernt ist, um auf das Signal zu reagieren oder ob sie steht. Wenn einer der beiden Fälle zutrifft, so prüft sie, ob das Signal bereits in einem vorherigen Schritt gesehen und registriert wurde. Falls Nein, so wird registriert, dass die Tram ein Stop-Signal erkannt hat.

Nach dieser Registration eines Signals, wird geprüft, ob auch auf ein weiteres, weiter entferntes Signal geachtet werden muss. Falls Ja, so wird die Position zu Beginn des neues Gleisstücks errechnet und „checkStop()“ mit dieser neuen Position und dem verbleibenden Rest der Vorausschau gestartet.

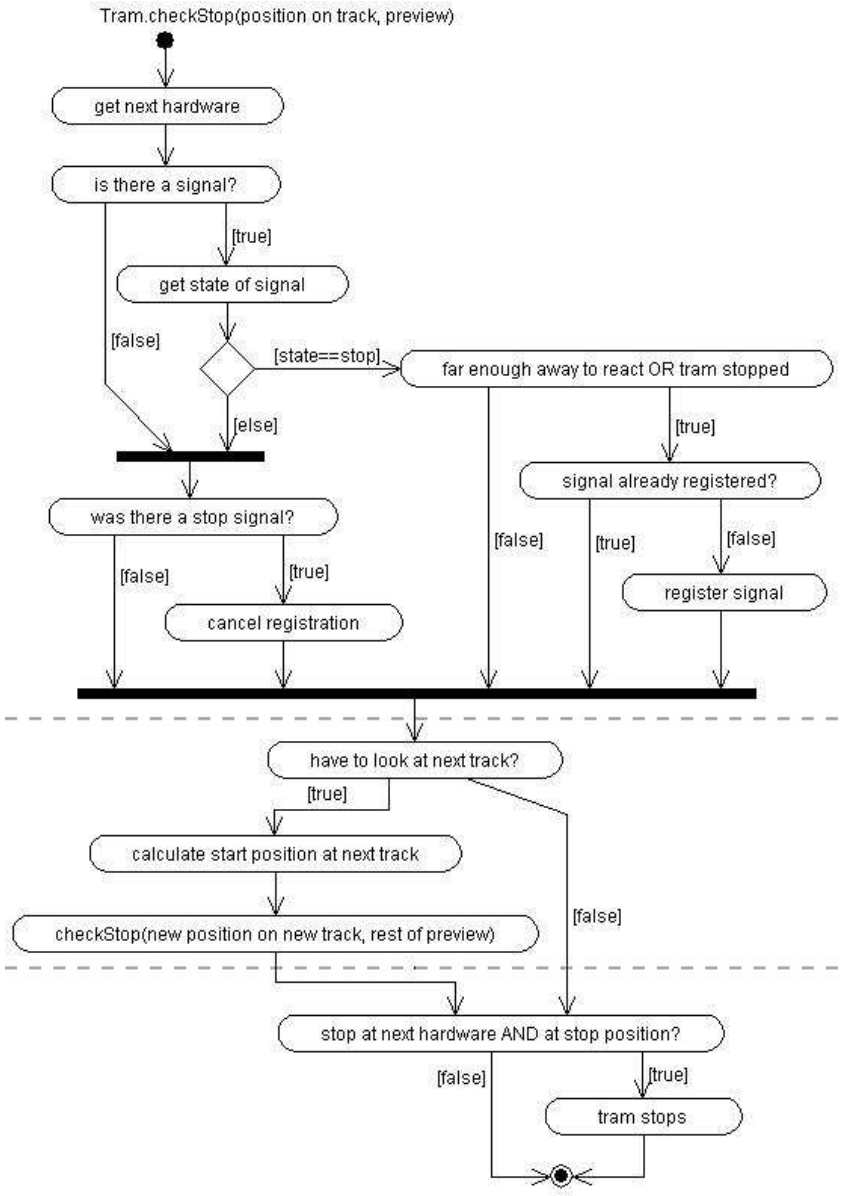


Abbildung 7.70: Aktivitätsdiagramm: Tram.checkStop()

Nachdem alle möglichen Stop-Signale registriert sind, wird geprüft, ob an dem nächsten Hardware-Element ein Stop-Signal steht und ob sich die Tram unmittelbar vor diesem Signal befindet. Falls Ja, stoppt die Tram.

Tram.calculatePosition() „calculatePosition()“ errechnet auf Grundlage der alten Position, die als Parameter übergeben wird und in die die neue Position ein-

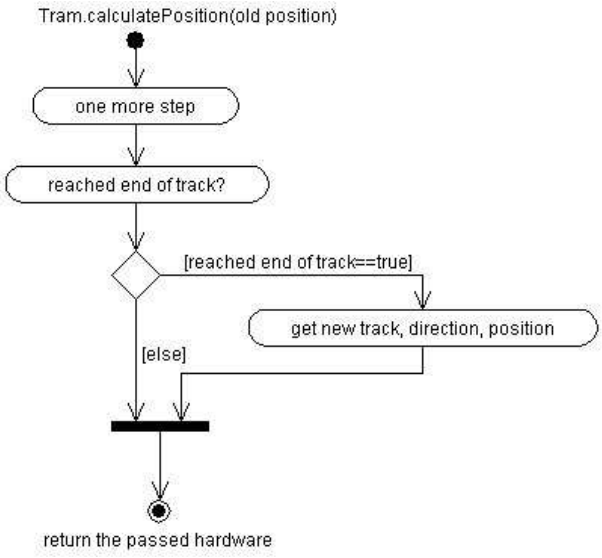


Abbildung 7.71: Aktivitätsdiagramm: Tram.calculatePosition()

getragen wird, die nächste Position. Diese Position untergliedert sich in Gleisstück, Richtung und ‚Position auf dem Gleis'. Der Algorithmus läuft wie folgt ab: zuerst wird abhängig von der Richtung auf dem aktuellen Gleisstück ein Schritt weiter nach vorne gerückt. Falls durch diesen Schritt das Ende des Gleisstücks erreicht wurde, muss auf das nächste gewechselt werden, in dem das neue Gleisstück, die Richtung auf dem neuen Gleis und die ‚Position auf dem Gleis' berechnet wird. Zurückgegeben wird die überfahrene Hardware oder der Wert „null".

Tram.driveInto() „driveInto()" bekommt als Parameter die Position, an der geprüft werden soll, ob die Tram auf eine andere Tram auffahren würde. Alle Straßenbahnen im Gleisnetz werden geholt und einzeln überprüft. Da es keinen Sinn macht, dass eine Tram auf sich selbst auffährt, wird dieser Fall nicht überprüft. Falls eine andere Straßenbahn gefunden wurde, so wird geprüft, ob das Ende der dieser Straßenbahn auf der zu prüfenden Position ist. Falls Ja, stoppt die Straßenbahn. Falls Nein, wird die Prüfung mit den restlichen Straßenbahnen fortgesetzt.

Tram.checkHeadCollision() „checkHeadCollision()" prüft, ob eine Straßenbahn mit einer anderen frontal zusammengestoßen ist. Dazu werden alle anderen Straßenbahnen wie bei „driveInto()" überprüft. Jedoch wird hier geprüft, ob sich der Kopf einer anderen Straßenbahn an der gleichen Position befindet wie der Kopf der eigenen Tram. Falls Ja, ist ein Zusammenstoß passiert und dem Simulator wird Bescheid gegeben. Danach ist die Methode beendet.

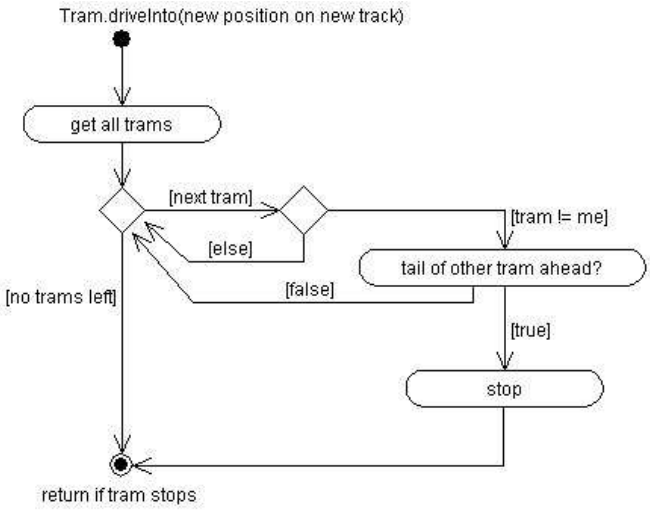


Abbildung 7.72: Aktivitätsdiagramm: Tram.driveInto()

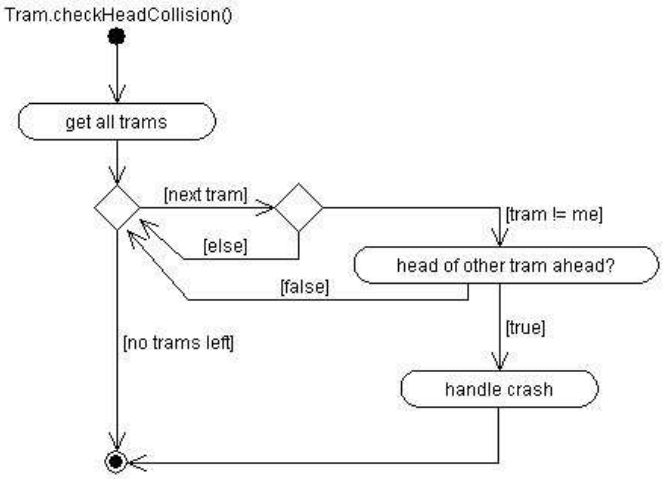


Abbildung 7.73: Aktivitätsdiagramm: Tram.checkHeadCollision()

Tram.checkCollision() Diese Funktion prüft, ob es eine andere Kollision außer dem Frontal-Zusammenstoß gibt. Dies ist nur möglich, wenn die Tram auf ein neues Gleisstück gewechselt hat. Übergeben wird die Hardware, die gerade überfahren wurde. Wenn es sich dabei um eine Weiche oder ein Kreuzung handelt, beginnt der Prüfvorgang. Wenn das Hardware-Element als besetzt gekennzeichnet ist, so bedeutet dass, das sich eine anderen Straßenbahn darauf befindet (oder darauf befand, aber sich im selben Simulatorzyklus bereits davon wegbewegt hat). Dann ist es zu einem Crash gekommen und dies wird dem Simulator gemeldet.

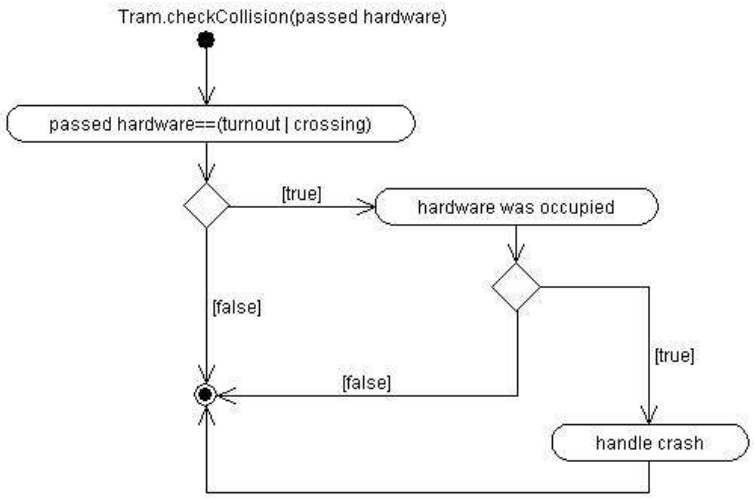


Abbildung 7.74: Aktivitätsdiagramm: Tram.checkCollision()

Tram.doFinish() „doFinish()“ führt Arbeiten aus, die durch die Fortbewegung der Straßenbahn hervorgerufen werden. Als Parameter bekommt diese Methoden die von Kopf und Ende überfahrenen Hardware-Elemente und das alte Gleisstück, auf dem der Kopf war. „doFinish()“ gliedert sich in zwei Elemente, die ähnlich ablaufen: einmal der Bereich für den Kopf (Aktionen, die durch den Kopf hervorgerufen wurden) und dann der Abschnitt für das Ende der Tram (Aktion, die auftreten, wenn die Bahn etwas verlässt).

Hat der Kopf ein Hardwareelement überfahren, so wird diese Hardware als ‚besetzt‘ markiert (wichtig für die Kollisionsüberprüfung, s. „checkCollision“). Handelt es sich dabei um eine Weiche oder Sensor, so merkt sich die Tram, dass sie diese überfahren hat (da Sensoren auslösen und Weichen passiv gestellt werden können). Wurde ein Sensor überfahren, so muss er aktiviert werden. Dabei wird das alte Gleiselement übergeben, damit Richtungssensoren feststellen können, in welcher Richtung sie überfahren werden.

Wurde ein Gleisstück von der Tram vollständig verlassen, so muss die überfahrene Hardware als nicht mehr ‚besetzt‘ gekennzeichnet werden. Dabei wird allerdings noch vermerkt, dass sie erst in diesem Zyklus verlassen wurde, um die Kollisionsabfrage nicht vom Zufall, welche Bahn zuerst in einem Zyklus bewegt wird, abhängig zu machen. War das überfahrene Element ein Sensor, so wird dieser deaktiviert. Damit ist die Methode beendet.

7.8.4.4.3 Initialisierung einer Simulation Bevor eine Simulation gestartet werden kann, ist es notwendig, zuerst grundlegende, initiale Daten festzuhalten. Diese Daten dienen der Spezifikation einer bestimmten Situation, die mit dem Simulator nachgestellt

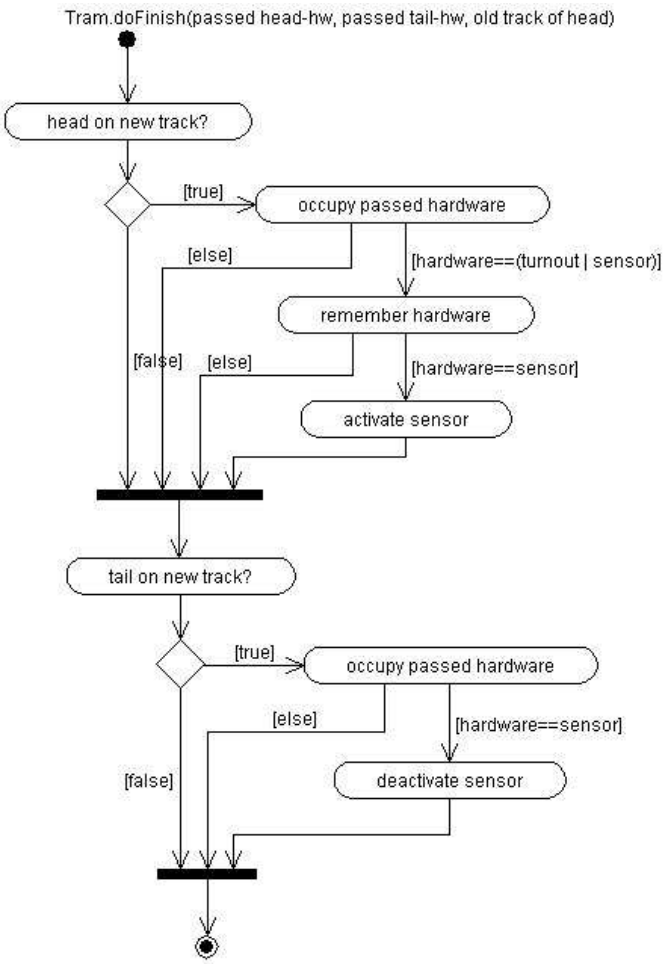


Abbildung 7.75: Aktivitätsdiagramm: Tram.doFinish()

werden soll. Zur Eingabe dieser Initial-Daten eignet sich eine graphische Oberfläche am besten. In der Visualisierung des Gleisnetzes erkennt der Benutzer u. a. Weichen, Sensoren und Signale. Diese drei Hardware-Elemente plus die Straßenbahn(en) können vom Benutzer gesetzt/gestellt werden. Die Spezifikation einer Simulation verläuft entlang der eben genannten Elemente:

- Straßenbahn(en) setzen – *Set Trams*
  Das Setzen von Bahnen erfolgt in zwei Schritten. Zunächst wird ein Dialog gezeigt, in dem der Nutzer die gewünschte Anzahl der Bahnen festlegen kann, in einem zweiten Dialog kann er dann nähere Angaben zu den Bahnen eingeben.

  Bei einer Straßenbahn gibt es folgende Spezifikationsmöglichkeiten:

  – Route Die Route ist die Strecke, die die Straßenbahn entlang fahren soll. Es

kann eine Route aus der Liste aller Routen ausgewählt werden. Nach Start der Simulation wird die Bahn an einer Einfahrt in das Gleisnetz eintreten, die zu einem RR-Sensor führt, der die gewünschte Route anfordern kann. (Dazu ist es wichtig, dass sich zwischen Einfahrt und RR-Sensor keine Weiche befindet, da sonst die Bahn nicht garantiert bis zum RR-Sensor gelangt, da für die Bahn kein Einfluss auf Stellung der Weiche besteht. Aus gleichem Grund darf kein Signal dazwischen vorhanden sein.) Hat die Bahn die Route erfolgreich absolviert und das Netz verlassen, wird sie kurz darauf wieder an bereits genannter Einfahrt erscheinen und ihre Route erneut absolvieren. Es handelt sich also hier um Bahnen, die ihre Routen immer wiederholt befahren.

- Länge
  Die Länge gibt an, wie lang die Straßenbahn sein soll. Es kann eine Länge aus einer Liste ausgewählt werden.
- maximale Geschwindigkeit
  Die maximale Geschwindigkeit gibt an, wie schnell diese Straßenbahn maximal fahren kann. Es kann eine Geschwindigkeit aus einer Liste ausgewählt werden.
- „May fail“
  Durch Aktivieren dieser Checkbox setzt der Nutzer die Option, dass eine Bahn während des Betriebes ausfallen kann. Sie bleibt dann irgendwann stehen, an welchem Ort dies geschieht, ist nicht vorhersehbar.
- „Random behavior“
  Eventuell könnte der Fahrer dieser Bahn sich zum Missachten von Signalen entschließen...

Die Zuweisung von Namen zu einer Straßenbahn ist nicht möglich. Die Bahnen werden automatisch vom System durchnummeriert.

Es ist möglich, dass mehrere Straßenbahnen die gleiche Spezifikation, insbesondere die gleiche Route, haben. In diesem Falle werden die beiden Bahnen nacheinander in die Route einfahren.

Mittels des Knopfs „OK“ wird der Initialisierungsschritt der Straßenbahnen abgeschlossen und zum nächsten Schritt übergegangen.

- Weichen stellen – *Set Turnouts*
  Im nächsten Spezifikationsschritt werden die Weichen gestellt. Es erscheint eine Liste aller Weichen, die im Gleisnetz vorhanden sind.

  Bei Weichen sind folgende Einstellungen möglich:

  - Stellung
    Die Stellung der Weiche, in der sie zu Beginn der gezielten Simulation sein

wird. Aus einer Liste kann eine Stellung ausgewählt werden. Die Elemente dieser Liste sind abhängig von dem Typ der Weiche. So ist bei einer LR-Weiche nicht die Stellung „Straight“ möglich.

- „May fail“
  Durch Aktivieren dieser Checkbox setzt der Nutzer die Option, dass eine Weiche während des Betriebes ausfallen kann. Sie wird dann keine Schaltbefehle mehr ausführen. Wann der Ausfall eintritt, ist unvorhersehbar.
- „Random behavior“
  Die Weiche wird sich potentiell eigenmächtig umstellen, wenn diese Option aktiviert ist.

Mittels des Knopfs „OK“ wird der Initialisierungsschritt der Weichen abgeschlossen und zum nächsten Schritt übergegangen.

- Sensoren stellen – *Set Sensors*
  Die Möglichkeit, Sensoren zu stellen, wird im dritten Initialisierungsschritt gegeben. Für Sensoren gibt es lediglich die verschiedenen Defekte zur Auswahl:
  - „May fail“
    So eingestellte Sensoren werden irgendwann keine Meldungen mehr geben.
  - „Random behavior“
    Diese Option bewirkt, dass ein Sensor ggf. Rückmeldungen gibt, wenn keine Bahn anwesend ist.

  Mittels des Knopfs „OK“ wird der Initialisierungsschritt der Sensoren abgeschlossen und zum nächsten Schritt übergegangen.

- Signale stellen – *Set Signals*
  Im letzten Schritt der Spezifikation des gezielten Testens werden die Signale gestellt. Alle Signale, die im zugrunde liegenden Gleisnetz enthalten sind, werden aufgelistet. Optionen sind:
  - Stellung
    Die Stellung gibt an, was das Signal zu Beginn der Simulation anzeigen soll. Aus einer Liste kann die gewünschte Stellung ausgewählt werden. Die Elemente der Liste sind abhängig vom Typ des Signals, d. h. es kann nur eine Stellung ausgewählt werden, die das Signal auch annehmen kann.
  - RR
    Mit bis zu drei Checkboxen, je nach Vorhandensein der RR-Anzeigen am Signal, können diese RR-Anzeigen initial ein- oder ausgeschaltet werden.

  - „May fail“
    Durch Aktivieren dieser Checkbox setzt der Nutzer die Option, dass ein Signal während des Betriebes ausfallen kann. Es wird dann keine Schaltbefehle mehr ausführen. Wann der Ausfall eintritt, ist unvorhersehbar.
  - „Random behavior“
    Das Signal wird sich potentiell eigenmächtig umstellen, wenn diese Option aktiviert ist.

  Mittels des Knopfs „OK“ wird der Initialisierungsschritt der Signale und der gesamte Initialisierungsprozess abgeschlossen.

- Random Simulation
  Der Nutzer hat auch die Möglichkeit, auf Einstellungen fast vollständig zu verzichten. Er muss dann lediglich eine Abfrage, die für die Anzahl der Bahnen, absolvieren. Die Bahnen werden dann zufällige Routen wählen. Signale und Weichen bleiben dann im anfänglichen „undefined“-Zustand, bis sie vom SI erstmals geschaltet werden. (Gelangt eine Bahn an ein im undefinierten Zustand stehendes Signal, wird sie vorbeifahren oder halten. An einer undefinierten Weiche wird sie entgleisen.)

7.8.4.4.4 Simulation von fehlerhaft agierendem Gleisnetz Eine Teilaufgabe des Simulators im Projekt TRACS ist das Simulieren von Situationen, in denen ein oder mehrere Sensoren, Weichen, Signale oder Straßenbahnen falsch oder gar nicht reagieren. (vgl. Kapitel 7.8.2.2.2 auf Seite 318). Diese Testmöglichkeit wird im Simulator dadurch realisiert, dass man wie bereits oben beschrieben, bei der Konfiguration der Simulation bestimmte Elemente als ausfallgefährdet bestimmen kann. An dieser Stelle sollen noch einmal die möglichen Fehlersituationen zusammengefasst werden::

- *Tram* – die ausgewählten Straßenbahnen können folgende fehlerhafte Verhaltensweisen simulieren:
  - bleibt irgendwo liegen
    Die Straßenbahn wird in Kürze anhalten und sich dann nicht mehr von der Stelle bewegen. Dies simuliert den Fall, dass eine Straßenbahn aufgrund eines technischen Defekts oder anderer äußerer Umstände im Gleisnetz stehen bleibt.
  - beachtet kein Signal mehr
    Dies ist ein Fehlerfall, bei dem der Steuerinterpreter einen dadurch verursachten Unfall nicht verhindern kann und auch nicht muss, denn im Schienenverkehr wird davon ausgegangen, dass der Straßenbahnfahrer sich an die Signale hält. Dennoch kann man dadurch testen, ob der Steuerinterpreter sein

Möglichstes tut, um Unfälle, die aufgrund eines ignorierten Signals entstehen, zu vermeiden.

- *Weiche* – die ausgewählten Weichen können folgende fehlerhafte Verhaltensweisen simulieren:
  - Ausfall
    In diesem Fehlerfall reagiert die Weiche nicht mehr auf Befehle des Steuerinterpreters und bleibt in ihrer aktuellen Lage stehen – nur das befahren von der passiven Seite aus kann dann noch die Weiche (passiv) stellen. Der Steuerinterpreter muss dies dann erkennen.
  - eigenmächtige Schaltungen
    In diesem Fehlerfall wird die Weiche sich spontan in beliebige Stellungen umschalten. Wann und wie oft sie schaltet, wird im Programmlauf zufällig bestimmt. Auch hier muss dann der SI reagieren.
- *Sensor* – die ausgewählten Sensoren können folgende fehlerhafte Verhaltensweisen simulieren:
  - Ausfall
    Dieser Fehlerfall entspricht dem ersten Fehlerfall der Weiche – der Sensor gibt keine Rückmeldungen mehr. Dies ist vom SI allerdings nicht so leicht zu erkennen, da er die Signale von den Sensoren ja nicht mit seinen eigenen Anweisungen vergleichen kann, wie es bei der Weiche der Fall war.
  - liefert falsches Signal
    Wenn diese Option ausgewählt wird, dann reagiert der Sensor unkontrolliert und liefert falsche Rückmeldungen.
- *Signal* – die ausgewählten Signale können folgende fehlerhafte Verhaltensweisen simulieren:
  - Ausfall
    Dies bedeutet im übertragenen Sinn das gleiche wie ein Ausfall der Weiche, nur dass hier das Signal nicht mehr schaltet. Abhängig vom Zustand, in der das Signal hängen geblieben ist, muss der SI dann auf verschiedenste Weise das Gleisnetz wieder ‚sicher‘ machen.
  - eigenmächtige Schaltungen
    In diesem Fehlerfall wird das Signal sich spontan umschalten. Wann und wie oft es schaltet, wird im Programmlauf zufällig bestimmt. Auch hier muss dann der SI reagieren.

Zu beachten ist, dass die Elemente durch Anwahl der oben erklärten Optionen zunächst in einen internen Zustand übergehen, der besagt, dass der angewählte Fehler auftreten *kann*. Solche als „ausfallgefährdet“ bestimmte Elemente werden während des Programmlaufs dann an zufälligen Zeitpunkten in das tatsächliche Ausfallverhalten übergehen. Dagegen werden Elemente, bei denen keine Fehleroption angewählt wurde, ihr korrektes Verhalten im gesamten Simulationslauf beibehalten.
Weiterhin soll die Möglichkeit geboten werden, auch die Elemente, die ausfallen sollen, zufällig auswählen zu lassen. Dies ist dann auch die einzige Möglichkeit, Ausfälle in Simulationen vorkommen zu lassen, die über oben beschriebene Zufallskonfiguration gestartet wurden, da bei dieser ja keine Startkonfiguration der einzelnen Elemente erfolgt. Hierfür wird im Hauptmenü des Simulators eine ein- und ausschaltbare Option „Random Defects“ geboten, die bei laufendem Betrieb beliebig betätigt werden kann. Ist sie eingeschaltet, können beliebige Hardwareelemente spontan in den „ausfallgefährdeten“ Zustand wechseln. (Wichtig: Wenn ein Element erst einmal in diesem Zustand ist, kommt es nicht wieder zurück zum Normalverhalten - wenn man also einmal während des Simulationslaufs die Option eingeschaltet hatte, können sich Elemente „infiziert“ haben, aber erst nach Abschalten der Option der Ausfall tatsächlich stattfinden, da der tatsächliche „Ausbruch“ wiederum an einem zufälligen Zeitpunkt erfolgt.)

7.8.4.5 Visualisierung

Die Visualisierung der Simulation dient dem leichteren Verständnis, wie die Simulation abläuft. Damit die Zeichen-Routinen den Simulator nicht abbremsen, läuft die Visualisierung in einem eigenen Thread. Die graphische Darstellung soll korrekt sein – aber die Formschönheit spielt eine untergeordnete Rolle (vgl. Anforderung s. 7.8.2.3 auf Seite 318). Die zu visualisierenden Elemente sind Gleise, Weichen, Sensoren, Signal und Straßenbahnen. Bei den Hardware-Elementen Weiche, Sensor und Signal soll außerdem die Stellung und die Information des Typs erkennbar sein. Markierungen werden nicht extra visualisiert, da sie ein funktionsloser Teil des Gleisnetzes sind, der bereits durch die Gleise visualisiert wird.
Bei der Visualisierung handelt es sich um eine Darstellung in 2D in der Draufsicht. Es werden keine Grafiken benutzt, sondern alle Elemente werden gezeichnet. Die Visualisierung der einzelnen Elemente wird einfach und eindeutig gehalten. Da bei der Darstellung von Stellungen und Zuständen symbolisch Farben, Symbole und Abkürzungen verwendet werden, wird es eine Hilfe geben, die jederzeit aufgerufen werden kann und in der der Zustand der Gleiselemente in Textform ausgegeben wird. Die zugrunde liegenden Koordinaten werden entsprechend der Fenstergröße skaliert. Bei Veränderung der Fenstergröße wird die Visualisierung durch Skalierung angepasst. Eine bestimmte Mindestgröße für das Fenster sowie ein geringer Abstand zwischen den äußeren Gleisen und dem Rand werden vorausgesetzt.
Nun folgt die Darstellung der Reihenfolge, in welcher die verschiedenen Elemente visua-

lisiert werden und wie welches Element genau dargestellt werden soll, um den Anforderungen zu genügen.

7.8.4.5.1 Welt- → Pixelkoordinaten Die (Welt-)Koordinaten, welche in der TND-Datei niedergeschrieben sind und welche anschließend in die Klassenstruktur übernommen werden, müssen, bevor etwas visualisiert werden kann, in Pixelkoordinaten umgerechnet werden. Für die Umrechnung benötigt man zuerst einmal folgende Daten:

- die kleinste / größte x-Weltkoordinate (*xmin* / *xmax*)
- die kleinste / größte y-Weltkoordinate (*ymin* / *ymax*)
- die kleinste / größte x-Pixelkoordinate (*Pxmin* / *Pxmax*)
- die kleinste / größte y-Pixelkoordinate (*Pymin* / *Pymax*)

Diese Werte werden berechnet, indem einmal (nach dem Einlesen einer TND-Datei und vor der ersten Visualisierung) über alle Hardwareelemente nach diesen Werten gesucht wird. Die Pixelkoordinaten entsprechen denen vom Fenster, indem das Gleisnetz angezeigt werden soll.
Aus diesen Werten kann man nun die Verhältnisse berechnen zwischen Welt- und Pixelkoordinaten (*xMultiplier* und *yMultiplier*).

$$xMultiplier = (Pxmax - Pxmin)/(xmax - xmin) \quad (7.1)$$
$$yMultiplier = (Pymax - Pymin)/(ymax - ymin) \quad (7.2)$$

Wenn man zum einen alles im Bild haben möchte und zum anderen keine Verzerrung erlaubt, dann nimmt man für beide Koordinaten den kleinsten der beiden *Multiplier*:

$$Multiplier = min(xMultiplier, yMultiplier) \quad (7.3)$$

Mit Hilfe von diesem *Multiplier* (der bei einer Änderung der Fenstergröße angepasst werden muss) kann man nun die Koordinaten umrechnen und zwar folgendermaßen:

$$pixelCoord = (weltCoord - xmin) * Multiplier \quad (7.4)$$

Damit wir keine Elemente direkt am Rand eines Fensters zeichnen müssen, wurde beschlossen, ein wenig Platz zwischen dem Rand und der Zeichenfläche zu lassen. Die Dicke des Randes in Pixel legen wir unter der Konstanten *BORDERSIZE* fest. Für die Einbeziehung dieses Randes in die Formeln muss folgendes gemacht werden:

1. *Pxmin* und *Pxmax* müssen angepasst werden
2. *x*- und *y-Multiplier* müssen neu berechnet und *Multiplier* neu ausgewählt werden

3. Formel 7.4 muss geändert werden

Die nötigen Änderungen sind in den Formeln 7.5 bis 7.7 zusammengefasst:

$$Pxmin = Pxmin + BORDERSIZE \tag{7.5}$$
$$Pxmax = Pxmax - BORDERSIZE \tag{7.6}$$
$$pixelCoord = BORDERSIZE + ((weltCoord - xmin) * multiplier) \tag{7.7}$$

Mit diesen Formeln können wir zu jeder Zeit die zugehörigen Pixelkoordinaten zu den Weltkoordinaten ausrechnen. Dabei müssen *xmin*, *xmax*, *ymin* und *ymax* nur einmal am Anfang ausgerechnet werden, da keine weiteren Elemente hinzukommen und *Pxmin*, *Pxmax*, *Pymin* und *Pymax* müssen immer nur dann neu ausgerechnet werden, wenn die Fenstergröße geändert wird.

7.8.4.5.2 Zeichenreihenfolge Damit die Visualisierung der Simulation für das menschliche Auge nachvollziehbar dargestellt wird, ist es wichtig, auf die richtige Zeichenreihenfolge zu achten. Wenn z. B. die Gleise erst nach der Straßenbahn gezeichnet werden, so verwirrt das und erweckt den Anschein, als ob die Straßenbahn unterhalb der Gleise fahren würde. Die Reihenfolge der einzelnen Visualisierungen wurde wie folgt festgelegt:

- Weichen (verdicken die Gleise)
- Sensoren (liegen auf Gleisen)
- Kreuzungen (verdicken die Gleise)
- Straßenbahnen (fahren auf Gleisen, Weichen, Kreuzungen, Sensoren)
- Gleise (bilden die Grundlage)
- Signale (stehen neben Hardware-Elementen)

7.8.4.5.3 Zeichenmethoden Anhand der Zeichenreihenfolge werden nun die verschiedenen Darstellungen aller Elemente vorgestellt.

- Weichen
  Weichen und ihre Stellung werden dadurch visualisiert, dass die Gleise, die in der aktuellen Stellung überfahren werden können, dicker gezeichnet werden. Wenn sich eine Weiche in einem Umschaltvorgang befindet, dann wird nur das Gleisstück der Stammseite (*stem*) hervorgehoben.

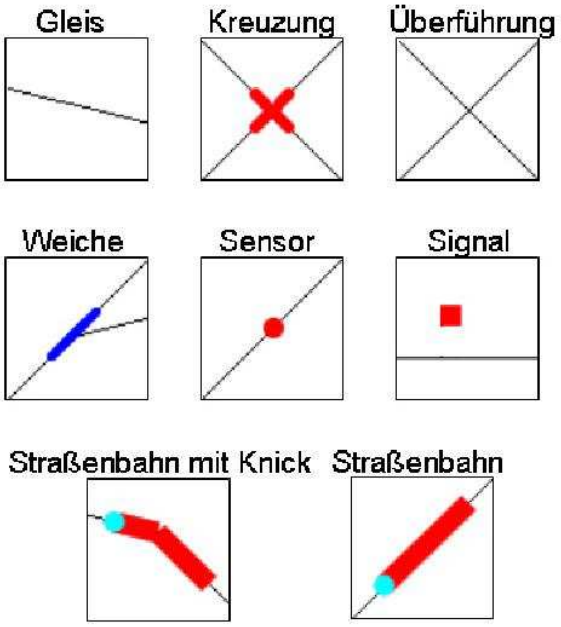


Abbildung 7.76: Visualisierung der Gleiselemente

- Sensoren
  Sensoren befinden sich auf einem Gleisstück und werden daher farbig mit einer gewissen Ausdehnung gezeichnet. Sie werden durch einen kolorierten, ausgefüllten Kreis visualisiert. Die Farbe zeigt abhängig vom Typ des Sensors den Zustand an. *ToggleSensoren* haben zwei Farben für die beiden Toggle-Stellungen. *Zustandssensoren* sind zwei andere Farbtöne zugewiesen für die Visualisierung des Zustands. *RouteRequestSensoren* stellen ihren Zustand, ob sie ausgelöst sind oder nicht, ebenfalls durch Farben wie die Zustandssensoren dar, wobei im Zeitpunkt des Auslösens neben der Zeichung des Sensors die Bezeichnung der angeforderten Route erscheint.

- Kreuzungen
  Kreuzungen, auf denen es zu einem Zusammenstoß kommen kann, müssen sich unterscheiden von Überführungen, bei denen das eine Gleis über eine Brücke über das andere Gleis hinüber führt. Eine Kreuzung wird dadurch angezeigt, dass die Gleise in Kreuzungsmitte dicker gezeichnet werden. Im Gegensatz dazu wird eine

Überführung nicht extra gekennzeichnet (s. Abbildung 7.76 auf der vorherigen Seite, erste Zeile).

- Straßenbahnen
  Straßenbahnen werden mit Form eines Rechtecks gezeichnet. Der Kopf wird durch eine andere Farbe und durch eine runde Form gekennzeichnet. Überfährt die Straßenbahn gerade ein Hardware-Element, so knickt sie in der Mitte (s. Abbildung 7.76 auf der vorherigen Seite, vierte Zeile). Der Zwischenraum, der auf der Außenseite entsteht, wird nicht umständlich ausgefüllt, sondern bleibt erhalten aufgrund der untergeordneten Rolle der Formschönheit.

- Gleise
  Gleise werden durch dünne Linien visualisiert. Da ein Gleisstück keine Krümmung aufweisen kann, handelt es sich um eine einfach Gerade mit Start- und Endpunkt, die gezeichnet wird. Die Linie ist im Normalfall ohne Farbe (bzw. Farbe schwarz) gehalten. Zu beachten ist, dass Gleise nicht eigenständig gezeichnet werden, sondern zusammen mit den angrenzenden Hardware-Elementen: Immer, wenn eines der o.a. HW-Elemente gezeichnet wird, werden die angrenzenden Gleise jeweils in halber Länge mitgezeichnet (die fehlende zweite Hälfte wird ja dann mit dem dort angrenzenden HW-Element zusammen gezeichnet).

- Signale
  Signale stehen neben den Gleisen an ihren definierten Ortskoordinaten und werden durch quadratische Kästchen visualisiert. Dei der Symbolisierung der Zustände wird sich weitgehend an der Darstellung der echten Symbole orientiert. Demnach ist der waagerechte Balken das Haltsignal, der senkrechte ein „Go straight“, der von links unten schräg nach rechts oben führende Balken ein „Go right“ usw. Bei einem der Wartezustände (runder Punkt) wird zusätzlich durch den Buchstaben S, L oder R angegeben, welcher Wartezustand es ist. Die bis zu drei RR-Anzeigen werden durch Buchstaben unter dem Signal (L, S, R) angezeigt, wenn sie aktiv sind. Die Richtung, aus der ein Signal zu sehen ist, ist gekennzeichnet, in dem die Hintergrundfarbe des quadratischen Kästchens und die des auf das Signal zuführenden Gleises identsch ist und von der normalen Gleisfarbe schwarz abweicht.

- Weitere Informationen
  Jedes Hardware-Element kann angeklickt werden, es öffnet sich dann ein Fenster mit Informationen zu diesem Element.

7.8.4.6 Treiber zur Anbindung an den SI

Der gesamte Kommunikationsvorgang zwischen Steuerinterpreter und Simulator stellt sich in folgenden Schritten dar: Der Steuerinterpreter kommuniziert mit den Hardware-Treibern über ein Shared Memory und die Treiber ihrerseits kommunizieren mit dem

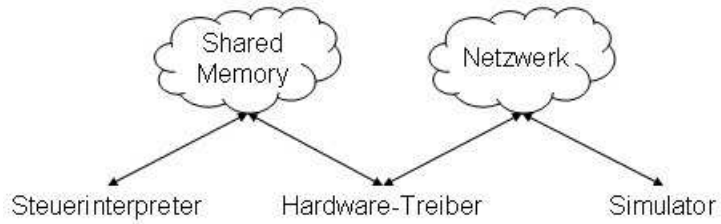


Abbildung 7.77: Kommunikation zwischen Steuerinterpreter, Treibern und Simulator

Simulator über Netzwerk.
Das Netzwerk ist dabei das Mittel der Wahl, da die Systeme Steuerinterpreter und Simulator nicht gleichzeitig auf demselben Rechner, sondern auf verschiedenen Rechnern laufen (Grund hierfür ist, dass der Steuerinterpreter möglichst unbehelligt von anderen Systemen laufen können soll). Vorgesehen ist, dass auf diesem Netzwerk die TCP/IP-Protokollfamilie eingesetzt werden soll, somit müssen beide Rechner diese Protokolle kennen. Besonders hohe Anforderungen an die Mindestbandbreite existieren nicht: Geht man von einer durchschnittlichen Nachrichtenlänge von 30 Zeichen aus, nimmt man ferner an, dass 100 Hardware-Elemente im Gleisnetz existieren und pro Sekunde für jedes HW-Element eine Nachricht gesendet wird, so kommt man auf 3000 Zeichen pro Sekunde – diese Anforderung wird selbst von heute gängigen Analogmodems überboten, von Ethernet ganz zu schweigen.
Im Nachfolgenden werden die verschiedenen Richtungen und das Format der Kommunikation beschrieben.

7.8.4.6.1 Eigenschaften der Kommunikation Die Kommunikation ist bidirektional: Vom Steuerinterpreter müssen über die Hardware-Treiber Anweisungen an die (simulierte) Hardware abgesetzt werden, in umgekehrter Richtung müssen die HW-Elemente Meldungen über ihren aktuellen Zustand ausgeben.

- Treiber $\Longrightarrow$ Simulator
  Der Steuerinterpreter sendet Anforderungen an Weichen und Signale, bestimmte Stellungen einzunehmen. Eine Nachricht enthält demnach

  – eine Kennzeichnung, dass es sich um eine Anforderung handelt
  – eine eindeutige Identifikation des angesprochenen Hardware-Elements
  – die Angabe der Stellung, in die geschaltet werden soll

  Da der Steuerinterpreter zu jeder beliebigen Zeit Anforderungen senden kann, muss auf Seiten des Simulators auch jederzeit eine Empfangseinheit bereitstehen. Hier ist ein eigener Thread als Server vorgesehen. Dieser Server schreibt die eingegangenen

Nachrichten in eine Liste, aus der der Simulator sie dann entnehmen und bearbeiten kann. Um die ggf. beim Zugriff auf die Liste auftretenden Nebenläufigkeitsprobleme zu beheben, darf der Zugriff jeweils nur einem Thread (Simulator oder Server) zum gleichen Zeitpunkt möglich sein. Vor dem Ablegen einer Nachricht in der Liste wird seitens des Servers bereits ein Korrektheitscheck vorgenommen, der Nachrichten, die nicht dem Vorgabeformat entsprechen, ohne Rückmeldung an den Steuerinterpreter verwirft. Aus Gründen der Einfachheit soll der Server jeweils nur eine Verbindung zur selben Zeit bearbeiten können, es werden also keine weiteren Threads erzeugt, die ggf. parallel Verbindungen annehmen. Der Server kann allerdings mehrere Verbindungen gleichzeitig offen halten, die dann eine nach der anderen abgefragt werden. Da die Nachrichten jeweils nur sehr kurz sind, erscheint uns dies als hinreichend.

- Simulator $\Longrightarrow$ Treiber
  Sämtliche Hardware-Elemente melden ihren Status an den Steuerinterpreter, wenn eine Statusänderung eingetreten ist. Eine Nachricht enthält

  – eine Kennzeichnung, dass es sich um eine Statusmeldung handelt
  – eine eindeutige Identifikation des betreffenden Hardware-Elements
  – die Angabe der Stellung, die eingenommen wurde (bei Weichen / Signalen) oder die Information über das Auslösen (bei Sensoren)

  Auch diese Sendeeinheit des Simulators wird als eigener Thread realisiert, damit der Simulator nicht durch einen ggf. länger dauernden Sendevorgang blockiert wird. Folglich ist auch hier ein Datenaustausch zwischen Simulator und Sender nötig, dieser erfolgt analog zu obigem Vorgehen ebenfalls über eine Liste. Der Simulator schreibt in diese Liste, der Sender prüft fortlaufend, ob sich etwas darin befindet und sendet die in der Liste befindlichen Meldungen an die Hardware-Treiber des Steuerinterpreters. Zur Nebenläufigkeit gilt das Gleiche wie oben.

7.8.4.6.2 Kommunikationsablauf und Nachrichtenformat Für die Kommunikation wurde ein Nachrichtenformat definiert, das im Folgenden beschrieben werden soll. Beginnen soll die Beschreibung jedoch mit dem Kommunikationsablauf:

- Jeder Treiber muss sich nach seinem Start beim Simulator anmelden. Die Anmeldenachricht enthält die Hardware-ID, für welche der Treiber zuständig ist.
- Die so entstandene Verbindung wird offengehalten und fortan in beide Richtungen benutzt.
- Der Simulator antwortet darauf mit einer ersten Statusmeldung, in der der Schaltzustand des Elements übermittelt wird.

- Im fortlaufenden Betrieb erfolgen lediglich Meldungen, wenn Änderungen am Zustand eingetreten sind, der neue Zustand überschreibt demnach den alten.
- Gleiches gilt für die Treiber: Meldungen werden nur abgesetzt, wenn ein neuer Zustand angefordert werden soll.
- Es findet kein Austausch von Empfangsbestätigungsmeldungen statt.
- Der SI erkennt die erfolgreiche Übertragung seiner Anforderungen daran, dass diese umgesetzt werden und entsprechende Statusmeldungen erfolgen.
- Fehlerhafte Meldungen, die nicht dem Nachrichtenformat entsprechen, sind beim Empfänger zu ignorieren, es findet keine Rückmeldung statt.
- Werden für ein Element gleichzeitig mehrere Zustände gemeldet, so ist die Reihenfolge, in der die Zustände aufgeführt werden, nicht vorgeschrieben.

Das Nachrichtenformat:

- Bei Anforderungen:
  `request` <`Element-ID`> <`Zustand`>, wobei `request` das Schlüsselwort für eine Anforderung darstellt, <`Element-ID`> der eindeutige Bezeichner des HW-Elements und <`Zustand`> der angeforderte Zustand ist. Die mögliche Zustände sind `left`, `straight`, `right`, `bent` (dieser Zustand kann nur bei SL- oder SR-Weichen bzw. Kreuzungsweichen angefordert werden, bei anderen führt Anforderung dieses Zustandes zum undefinierten Zustand), `quit` bei Weichen – bei Signalen sind es `quit`, `left`, `right`, `straight`, `stop`, `waitright`, `waitstraight`, `waitleft`, `rrrighton`, `rrlefton`, `rrstraighton`, `rrrightoff`, `rrleftoff`, `rrstraightoff`. Auch an Sensoren kann eine „Anforderung“ gesendet werden, die einizig mögliche ist hierbei `quit`. Das Senden einer quit-Anweisung führt dazu, dass der Simulator für dieses Element keine weiteren Rückmeldungen mehr erstattet, da er damit rechnen muss, dass der Treiber sich beendet hat.
- Bei Statusmeldungen:
  `status` <`Element-ID`> <`Zustände`>, wobei `status` das Schlüsselwort für eine Statusmeldung darstellt, <`Element-ID`> der eindeutige Bezeichner des HW-Elements und <`Zustände`> eine nicht-leere Menge von Zuständen ist. Mögliche Zustände sind `left`, `straight`, `right`, `undefined` bei Signalen und Weichen, bei Signalen kommen noch noch `stop`, `waitstraight`, `waitleft`, `waitright`, `rrstraighton`, `rrstraightoff`, `rrrighton`, `rrrightoff`, `rrlefton` und `rrleftoff` hinzu. Bei Sensoren sind die Zustände `dira` (Sensor wurde in Richtung A überfahren), `dirb` (analog Richtung B), `tramon` (Bahn anwesend), `tramoff` (keine Bahn anwesend), `toggleon` (Toggle-Bit gesetzt), `toggleoff` (Toggle-Bit nicht

gesetzt) und `rr` (Route-Request ausgelöst), wobei bei Letzterem auch noch die Routenbezeichnung nach einem Leerzeichen mitgeschickt wird.

- Initiale Meldung an den Simulator:
  `init <Element-ID>` (siehe Beschreibung der Initialisierung oben)

7.8.4.7 Korrektheit

Die Korrektheit des Simulators kann letzten Endes sicherlich nicht vollständig gewährleistet werden, da der Simulator als nicht sicherheitsrelevante Software nicht in den Verifikationsprozess des Projektes eingebunden ist (lediglich soll mit ihm die CAD-TND-Konvertierung geprüft werden) und daher kein formaler Korrektheitsbeweis oder Ähnliches durchgeführt wird. Dennoch muss natürlich auf möglichst korrektes Funktionieren Wert gelegt werden. Dies soll gewährleistet werden, indem sowohl bei der Entwicklung des Programms auf einige Aspekte zur korrekten Quellcodeerstellung geachtet werden soll, als auch nach der Fertigstellung des Quellcodes ausführliche Tests durchgeführt werden sollen. Im Zusammenspiel sollen diese Tests inklusive der darauf folgenden Korrekturen eine möglichst hohe Fehlerfreiheit des Simulators sicherstellen.

7.8.4.7.1 Fehlervermeidung während der Entwicklung Um von vornherein die Anzahl der Fehler im Quellcode möglichst gering zu halten, so wie es in den Anforderungen im Kapitel 7.8.2.5 auf Seite 320 gefordert wurde, wird auf eine übersichtliche Datenstruktur im Programm geachtet (s. S. 328) und eine Programmiersprache gewählt, die solche eine Struktur überhaupt ermöglicht (s. S. 325). Des Weiteren werden bei der Entwicklung des Simulators folgende Aspekte, welche die Korrektheit des Programms fördern sollen, berücksichtigt:

- Ausführliche Dokumentationen
  Die Dokumentationen im Quellcode ermöglichen einerseits mit Hilfe von Doxygen Dokumente zu erzeugen, die die Funktionsweise der einzelnen Klassen, Methoden und Variablen erklären. Andererseits enthält der Quellcode auch viele Kommentare, die nicht in den von Doxygen generierten Dokumenten auftauchen, da sie dem Programmierer helfen sollen, den Quellcode zu erweitern bzw. zu korrigieren. Dies ist wichtig, wenn Projektteilnehmer von TRACS am Quellcode arbeiten sollen, die ihn nicht selbst erstellt haben.

- Cross-Reviews
  Bei wichtigen, für die Korrektheit des Programms entscheidenden und komplizierten Methoden oder auch Klassen werden diese von einem Mitglied der Simulatorgruppe durchgesehen, dass nicht an der Entwicklung des Quellcodes beteiligt war. Durch Nachfragen und Hineindenken dieses hinzugekommenen Mitglieds sollen vor allem konzeptuelle Fehler gefunden werden, da diese auch bei mehrfachem

Durchsehen des Entwicklers im Allgemeinen nicht auffallen. Im dritten Projektsemester war diese Technik bei der zuletzt vorliegenden Gruppengröße leider nicht mehr einsetzbar.

7.8.4.7.2 Fehlersuche im Anschluss an die Fertigstellung Außer den Maßnahmen zur Fehlervermeidung während der Entwicklung werden für den Simulator auch anschließende Tests durchgeführt, die die korrekte Funktionsweise zumindest annähernd sicherstellen sollen. Dafür werden im Voraus Methoden geschrieben, die den aktuellen Zustand in übersichtlicher und gut lesbarer Form in eine Textdatei oder auf den Bildschirm schreiben. Getestet werden soll im Anschluss:

- Funktion einzelner Methoden
  Dies ist die unterste Ebene der Tests. Hier werden vor allem die komplizierteren Methoden ausführlich und sehr im Detail auf ihre Richtigkeit hin überprüft. Als Beispiel wären hier die Methoden zur Fortbewegung der Bahn zu nennen.

- Funktion der Module
  Als zu testende Module bieten sich die Kommunikationsschnittstelle (beide Richtungen einzeln), die Sensoren, die Weichen, die Signale und die Straßenbahnen an. Hierbei soll nun nicht mehr jede Methode einzeln getestet werden, sondern die Ergebnisse, die solch ein Modul produziert. Beispiele für solche Tests wären der Gesamtvorgang des Umsetzens von Schaltanforderungen, die korrekte Fahrt über Weichen und Kreuzungen etc.

- Zusammenspiel von Daten und Visualisierung
  An dieser Stelle soll eher stichprobenartig getestet werden, ob die angezeigten Informationen mit den abgespeicherten in der Datenstruktur übereinstimmen. Dies betrifft vor allem die Informationen, die nicht 1-zu-1 übertragen und visualisiert werden können, wie die Ortskoordinaten und die Zustände von Sensoren, Weichen und Signalen. Auch die Vorausschau der Bahnen (zum nächsten Signal) empfiehlt sich hier zu testen, da sie ein interner Teil ist, der visualisiert leicht zu überprüfen ist.

- Funktion des Simulators mit angeschlossenem SI
  Die allgemeinste Ebene der Tests ist das so genannte „Black-Box-Testen“ des gesamten Simulators. Hierzu wird ein Steuerinterpreter angeschlossen, und Situationen initialisiert, die dann durchlaufen werden können. Dazu müssen vorher die erwarteten Ergebnisse aufgeschrieben werden, die nach dem Testlauf dann mit den tatsächlich durch den Simulator protokollierten Ergebnissen verglichen werden. Auf diese Tests muss bisher leider verzichtet werden, da kein funktionsfähiger Steuerinterpreter vorliegt.

7.8.4.8 Echtzeitfähigkeit

Der Simulator soll ein Gleisnetz samt (re-)agierender Sensoren, Weichen und Signalen sowie der darauf fahrenden Straßenbahnen simulieren. Der finanzielle Rahmen des Projektes TRACS erlaubt keine Hardwareausstattung, um für jedes oben genannte Element eine eigene CPU abzustellen. Der gesamte Simulator läuft auf einer einzigen CPU (s. S. 371). Wir benutzen ein Konzept der sequentiellen Abarbeitung sämtlicher Hardwareelemente. Wie dieses Konzept aussieht und warum wir keine Threads zur parallelen Abarbeitung benutzen, wird im ersten Unterkapitel erläutert.
Damit die Simulation ein Echtzeitsystem korrekt simuliert, muss die im Simulator als vergangen berechnete Zeit mit der tatsächlich vergangenen Zeit übereinstimmen. Auf diesen Aspekt wird im darauf folgenden Kapitel eingegangen.
Das letzte Unterkapitel schließlich beleuchtet die Konzepte, die befolgt werden, damit die Rechenzeit pro Runde möglichst gering gehalten wird.

7.8.4.8.1 Threads vs. sequentielle Abarbeitung Wir haben uns für eine sequentielle Abarbeitung und gegen die Benutzung von einem Thread pro Hardwarelement entschieden, da die Verwendung der Rechenzeit der CPU auf diese Weise effektiver erscheint. Einzige Threads sind die beiden Kommunikationsschnittstellen (Sender und Empfänger), der Simulator und die GUI.
Bei der Verwendung von einem Thread pro Hardwarelement hätte man wie beim anderen Konzept auch keine Parallelität, da nie zwei Threads wirklich zeitgleich agieren, sondern die Wechsel der Threads lediglich sehr oft und zu nicht festgelegten Zeitpunkten passieren. Für das Beispielnetz in Abb. 7.78 auf der nächsten Seite bräuchte man mindestens 27 Threads für die Hardwarelemente, zusätzlich 1 Thread pro Straßenbahn, 2 Threads für die Schnittstelle zu den Treibern und die von Java™selbst erzeugten Threads für die grafische Oberfläche. Zwischen diesen Threads könnten enorme Nebenläufigkeitseffekte erwartet werden, die bei einer sequentiellen Abarbeitung völlig vermieden werden. Die Behebung dieser Effekte würde weitere Rechenleistung verschlingen, die man im anderen Fall nicht einsetzen muss. Prinzipiell müsste der Rechenaufwand für eine Runde bei beiden Verfahren zunächst einmal der Gleiche sein, da die Berechnungen die Gleichen sind, egal ob sie sequentiell oder quasiparallel durchgeführt werden. Bei Threads käme dann aber der Zusatzaufwand zur Bekämpfung der Nebenläufigkeitseffekte hinzu. Wir haben uns daher für das Konzept der sequentiellen Abarbeitung entschieden, welches im Folgenden näher erläutert wird.
Das Konzept der rundenbasierten sequentiellen Abarbeitung sieht vor, dass die Hardwarelemente in einer klar definierten Reihenfolge nacheinander bearbeitet werden. Jeder ausgelöste Sensor und jede überfahrene Kreuzung und Weiche werden dabei benachrichtigt (s. S. 338). Alle anderen Elemente werden in der Runde nicht beachtet. Eine komplette Runde ist in folgende Abschnitte unterteilt:

- Abfrage der Inputs vom SI

Eventuelle, von den Hardwaretreibern geschickte, neue Anweisungen vom SI werden abgefragt.

- Verarbeitung der Inputs
  Die erhaltenen Anweisungen werden an die einzelnen Signale und Weichen weitergegeben.

- Hardware-Elemente schalten Alle HW-Elemente werden aktualisiert, d.h. solche, bei denen eine Schaltung fällig ist, schalten um. Es erfolgen die nötigen Rückmeldungen an die Treiber.

- Bahnen fortbewegen
  Dies ist der Hauptschritt. Die Bahnen fahren, überfahrene Sensoren werden ausgelöst, von der passiven Seite aus befahrene Weichen werden gestellt und eventuelle Zusammenstöße von Straßenbahnen werden ermittelt.

- Veränderungen des Systemzustands der Schnittstelle melden
  Die Zustände von Sensoren und Weichen, die bei der Bewegung der Bahnen verändert wurden, werden an die Treiber gesendet.

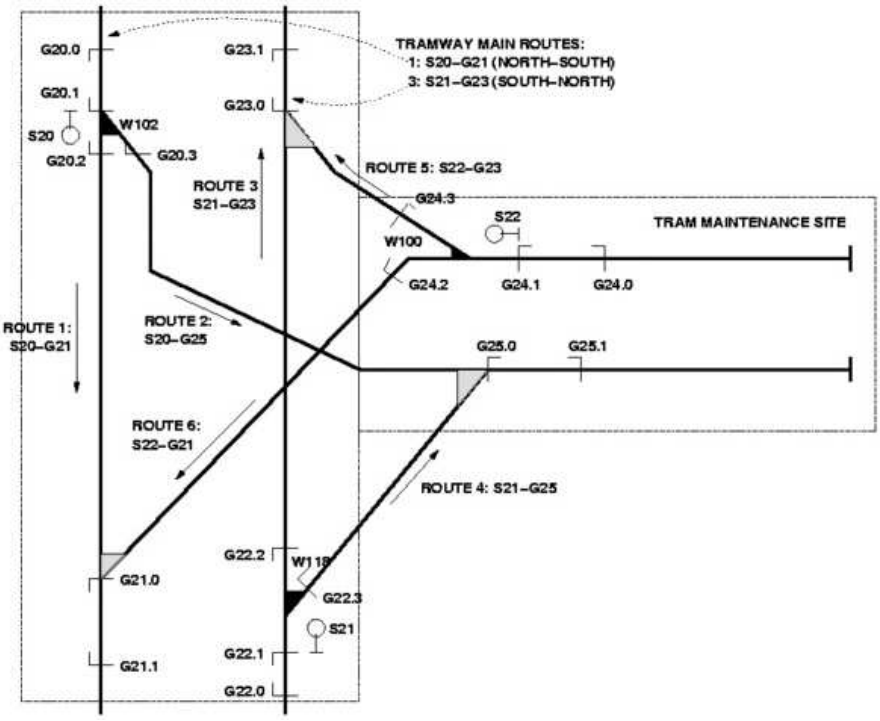


Abbildung 7.78: Beispielgrafik eines Gleisnetzes

7.8.4.8.2 Echtzeit vs. simulierte Zeit Für eine möglichst präzise Simulation eines Echtzeitsystems muss die reale Zeit (das ist die Zeit, die für den Anwender des Simulators vergeht) mit der als vergangen berechnete Zeit im Simulator übereinstimmen. Die im Computer vergangene Zeit setzt sich zusammen aus:

- Zeit für die Berechnung einer Runde der Simulation
- Zeit für die Verarbeitung von Nachrichten der Treiberschnittstelle
- Zeit für die Berechnung der Visualisierung
- Zeit für die arbeitenden Hintergrunddienste im System

Da es keine Anforderung des Simulators an ein Echtzeitbetriebssystem gibt, stellt gerade der letzte Punkt eine möglicherweise große und unberechenbare Größe dar. Durch einen Verzicht auf permanent der Echtzeit entsprechende Zustände im Simulator lässt sich jedoch eine andere ‚Art der Echtzeit' realisieren: es wird dafür gesorgt (soweit eben möglich), dass jeder Simulatorschritt gleich lang dauert, indem man nach Abschluss einer Runde eine bestimmte Zeit abwartet, bevor man die nächste Runde einleitet. Die Soll-Dauer einer Runde wird so groß gewählt, dass alle nötigen Berechnungen auf jeden Fall in dieser Zeit erfolgen können. Es wird die Dauer der Berechnungen gemessen und die Runde vervollständigt, indem die noch zur Soll-Dauer fehlende Zeit gewartet wird. Dieses Ziel wird durch das, in Pseudo-Code dargestellte, Konzept in Abb. 7.79 realisiert. Dabei wird nicht davon ausgegangen, dass die Systemuhr die absolut korrekte Uhrzeit

```
1    curTime = aktuelle Systemzeit;
2    if (prevTime + Simulator.STEP_DURATION > curTime) {
3        sleep (prevTime + Simulator.STEP_DURATION - curTime);
4    }
5    prevTime = aktuelle Systemzeit;
6    // ... start next round
```

Abbildung 7.79: Pseudo-Code für realitätsnahe Rundenbasiertheit

enthält – es wird allerdings davon ausgegangen, dass die während einer Simulation an der Systemuhr gemessenen Zeit der realen Zeit entspricht. Diese Voraussetzung wird von der verwendeten Hardware (s. S. 371) erfüllt.

7.8.4.8.3 Minimierung der Rechenzeit Ein Ziel der Simulation muss sein, die Bahnen in realitätsnahen Geschwindigkeiten fahren zu lassen. Die Geschwindigkeit der

Bahnen lässt sich über die Anzahl der Schritte, die sie pro Runde fährt, manipulieren (die Distanz pro Schritt ist konstant, die Dauer einer Runde, wie oben gezeigt, auch). Sichergestellt werden muss also, dass alle im Netz befindlichen Bahnen eine solche Zahl Schritte in einer Runde schaffen. Einige hierfür unterstützende Maßnahmen sind:

- Vermeidung von Berechnungen für die Visualisierung
- Vermeidung von Suchen in Mengen
- Vermeidung von Schreib- und Leseoperation
- Vermeidung von blockierenden Aufrufen

Zuerst einmal liegt bei der Form der Visualisierung keine Priorität. Die Fakten der Simulation sollen präsentiert werden – jedwede Schönheit der Simulation ist ein Bonus, der nicht auf Kosten der Genauigkeit der Simulation gehen soll. Allgemein ist die Ausgabe von Daten auf dem Bildschirm eine zeitaufwändige Sache – egal, ob es sich dabei um reinen Text oder Grafik handelt. Eine Schätzung, wie viel Rechenleistung die Bildschirmausgabe in Anspruch nimmt, liegt bei bis zu 30 % ([MDB03, S. 336]). Daher sollte die grafische Ausgabe so gering wie möglich gehalten und überflüssige Daten vermieden werden. Eine Optimierung der grafischen Ausgabe bietet der folgende Weg (der allerdings nicht beschritten worden ist): Bei der visuellen Darstellung der Simulation sollten nur die Veränderungen neu gezeichnet werden, statt das komplette Gleisnetz. Bewegt sich eine Straßenbahn auf dem Gleisnetz einen Schritt vorwärts, so wird nur der betreffende Bereich übermalt.
Das Suchen in einer Menge ist immer eine sehr zeitaufwändige Angelegenheit. Je nach Datenstruktur und Suchanfrage müssen viele Objekte gesucht und Membervariablen durchsucht werden. Hier bietet es sich an, die Geschwindigkeit beim Suchen durch eine erhöhte Verwendung von Speicher zu maximieren bzw. die Datenstruktur so aufzubauen, dass Suchen möglichst vermieden werden können (auf Kosten des Speicherverbrauchs).
Blockierende Aufrufe sind Methoden, die auf ein externes Ereignis warten (wie z. B. die Rückmeldung der Festplatte) und bis zum diesem Zeitpunkt nicht fortfahren können. In unserem Fall ist das Empfangen der SI-Anfragen blockierend, denn eine neue Anfrage soll jederzeit an den Simulator gesendet werden können. Dazu funktioniert die Simulator-Seite der Kommunikationsschnittstelle wie ein Postfach – alle ankommenden Aufträge werden gesammelt und können jederzeit vom Simulator abgerufen werden. Das Senden und Empfangen von Stell-Aufträgen muss also in einem eigenen Thread ablaufen, um ein Blockieren des ganzen Simulators zu vermeiden.

7.8.4.9 Hardware

Der Simulator wird für einen handelsüblichen Einprozessor-PC entwickelt. Dieser muss über eine Netzwerkanbindung verfügen, damit mit den Treibern und so letztendlich

dem Steuerinterpreter kommuniziert werden kann. (Die Alternative, ihn auf dem selben Rechner wie den SI auszuführen, stellt sich nicht.) Mindestanforderungen an Speicher und Prozessor konnten bisher nicht exakt ermittelt werden, es bleibt jedoch die Feststellung, dass auch die etwas älteren Rechner im Projektraum bisher keine Schwierigkeiten hatten, die benötigten Geschwindigkeiten zu liefern.

Silvia Graumann, Niklas Polke, Arne Stahlbock

7.8.5 Implementierung des Simulators

Dieses Kapitel beschäftigt sich mit der Implementierung in der Form, dass die wichtigsten und entscheidenden Stellen im Quellcode sowie diejenigen, die beispielhaft für eine Vielzahl von Stellen stehen, beschrieben und erläutert werden. Dieses Kapitel enthält bzw. beschreibt also nicht annähernd die Hälfte des gesamten Quellcodes – lediglich ausgewählte Beispiele beschreiben die Umsetzungen von Auszügen des Konzeptes. Folgendes wird beschrieben:

- der Einlesevorgang der TND anhand eines kleinen Beispiels
- Bewegung, Signalvorausschau, Auffahren und Kollisionsüberprüfung der Straßenbahn
- Visualisierung einer Straßenbahn
- Schnittstelle zu den Treibern – Übertragung von Anforderungen und Statusmeldungen
- Treiber: Beispiel Weichentreiber

7.8.5.1 Auszug aus einem TND-Einlesevogang

Das Einlesen einer TND-Datei ist keine triviale Angelegenheit, wie bereits im Kapitel 7.8.4.3 auf Seite 334 beschrieben wurde. Vor allem die Abbildung 7.63 auf Seite 335 gibt einen Eindruck darüber, wie viele ‚Stationen' durchlaufen werden müssen, bis die Informationen aus einer einzigen Zeile Text in der TND-Datei eingelesen und verarbeitet sind. An dieser Stelle wollen wir anhand eines Beispiels den Durchlauf und die darin enthaltenen Stationen erläutern.
Alle nun folgenden Codeauszüge sind nur Bruchteile der eigentlichen Dateien (lexer.def, parser.def, BestTramNetworkBuilderOfTheUNIverse.java) und teilweise auch vereinfacht dargestellt.
Die Zahlen am Anfang jeder Zeile sind nur für die Erklärung hinzugefügt worden – sie sind nicht Teil der originalen Dateien. Die vollständigen Dateien sind auf der dem Bericht beigefügten CD enthalten.

(Anmerkung: Dieses Beispiel stammt von September/Oktober 2004, inzwischen hat sich der Code der genannten Dateien wegen diverser TND-Änderungen ebenfalls verändert; dieses Beispiel wird aber hier beibehalten, da die Änderungen die Grammatik betreffen, nicht aber die Vorgehensweise, mit der der Simulator das Einlesen betreibt.)

7.8.5.1.1 Beispiel Unserer Beispieldatei im TND-Format liegt folgende Grammatik zugrunde, die in EBNF-Form aufgeschrieben ist:

```
01 <TND> ::= <DEFBLOCK> [<COORDSBLOCK>]
02
03 <DEFBLOCK> ::= "definitions" "{" {<TYPEDEF>} "}"
04 <TYPEDEF>  ::= <SENSDEF>
05 <SENSDEF>  ::= <SENSTYPE> ":" <SENSID> {"," <SENSID>} ";"
06 <SENSTYPE> ::= "tg-sensors"
07
08 <COORDBLOCK> ::= "coordinates" "{" {<COORDDEF>} "}"
09 <COORDDEF>   ::= <ELEMENTID> ":" "(" <COORD> "," <COORD>
                                    "," <COORD> ")" ";"
10 <ELEMENTID>  ::= <SENSID>
11 <COORD>      ::= ["-"] <DIGIT> {<DIGIT>}
```

Anmerkung: Die Zeilen 1, 4, 6 und 10 sind (teilweise sehr stark) vereinfacht worden, wodurch die meisten Zeilen der gesamten Grammatik entfallen. Eine gesamte TND-Datei kann aus sehr vielen Blöcken bestehen, doch unser Beispiel beschäftigt sich nur mit dem `DEFBLOCK` und dem `COORDBLOCK`. Diese Kombination wurde ausgewählt, da der letztere Block Bezug zu den Informationen aus dem ersten Block nimmt.
Nun folgt unsere Beispieldatei, die lediglich aus den gerade beschriebenen zwei Blöcken besteht. Im ersten wird ein Toggle-Sensor mit der ID `g003` definiert und im zweiten werden diesem Sensor Ortskoordinaten zugewiesen, nämlich $(1, -1, 0)$:

```
01 definitions {
02    tg-sensors: g003;
03 }
04
05 coordinates {
06    g003: (1,-1,0);
07 }
```

7.8.5.1.2 Lexer Im ersten Schritt des Einlesevorgangs wird diese Text-Datei vom Lexer eingelesen. Dieser liest jedes Zeichen einzeln ein und gibt die erkannten Token anschließend an den Parser weiter. Spezifiziert ist das Verhalten des Lexers in der Datei „lexer.def“ (diese wurde von der Compilergruppe erstellt und von der Simulatorgruppe

für JFlex angepasst) – aus dieser wird mit *JFlex* der Lexer („Lexer.java“) generiert (dabei wird die bei der Parsergenerierung erstellte Datei „Sym.java“ benötigt). Nun folgen auszugsweise einige Zeilen aus der Spezifikation des Lexers, die für die lexikalische Analyse der Beispieldatei notwendig sind:

```
01 "tg-sensors"     { return symbol(Sym.TGSENSORS, yytext()); }
02 "definitions"    { return symbol(Sym.DEFINITIONS); }
03 "coordinates"    { return symbol(Sym.COORDINATES); }
04
05 ":"              { return symbol(Sym.DOPPELPUNKT); }
06 ";"              { return symbol(Sym.SEMIKOLON); }
07 ","              { return symbol(Sym.KOMMA); }
08 "{"              { return symbol(Sym.GKA); }
09 "}"              { return symbol(Sym.GKZ); }
10 "("              { return symbol(Sym.KA); }
11 ")"              { return symbol(Sym.KZ); }
12
13 "g"{alnum}+      { return symbol(Sym.SENSID, yytext()); }
14 -?[:digit:]+     { return symbol(Sym.COORD,
                                    new Integer(yytext())); }
15
16 {WhiteSpace}     { /* Ignoriere sonstigen Whitespace. */ }
```

In den Zeilen 1-3 werden Symbole, stellvertretend für die erkannten Wörter, durch gereicht – Zeile 1 gibt zusätzlich das Wort als String mit, da es im Parser für die Weiterverarbeitung sogar noch an den BestTramNetworkBuilderOfTheUNIverse (BTNB) weitergegeben wird. In den Zeilen 5-11 werden die erkannten Zeichen als Symbole weitergegeben. In Zeile 13 wird eine ID erkannt – diese besteht aus einem g mit anschließenden Ziffern und Buchstaben. In der darauf folgenden Zeile wird eine Koordinate erkannt, welcher optional ein Minuszeichen voraus gestellt ist und daraufhin mindestens eine Ziffer folgt. Der eingelesene und erkannte String wird an dieser Stelle gleich in ein Integer gecastet und weitergegeben. Letztendlich wird sämtlicher „WhiteSpace“ (dazu gehören Leerzeichen, TabStops und Zeilenumbrüche) ignoriert.

7.8.5.1.3 Parser Nach dem Einlesen der Datei durch den Lexer erfolgt die Weiterverarbeitung der erkannten Token durch den Parser. Dieser muss nun schauen, dass er Produktionen findet, so dass die erkannten Token zu der vom Parser akzeptierten Grammatik passen. Ist dies nicht der Fall, so würde der Parser eine Fehlermeldung ausgeben. Der Parser selbst ist in der Datei „parser.def“ spezifiziert, welche ebenfalls von der Compilergruppe erstellt wurde und von der Simulatorgruppe, im Vergleich zu der Spezifikation des Lexers, gründlich verändert bzw. angepasst werden musste, teils weil

Differenzen zwischen CUP und yacc existeiern, teils weil die Art der Weiterverarbeitung sich zwischen Compiler und Simulator zu stark unterschieden. Mit Hilfe des bereits im Konzept genannten Tools *CUP* werden der eigentlich Parser („Parser.java“) sowie zusätzlich die Datei „Sym.java“ generiert – letztere ist für die Kommunikation zwischen Lexer und Parser notwendig. Nun folgt der für unser Beispiel benötigte Auszug aus der Spezifikation des Parsers:

```
01 tnd ::=          defblock coordblockopt
02                ;
03 defblock ::=     DEFINITIONS GKA GKZ
04                | DEFINITIONS GKA deflist GKZ
05                ;
06 deflist ::=      typedef
07                | typedef deflist
08                ;
09 typedef ::=      sensdef
10                ;
11 sensdef ::=      senstype:type DOPPELPUNKT senslist:list
                                              SEMIKOLON
12                  {: for (int i = 0; i < list.size(); i++) {
13                         BestTramNetworkBuilderOfTheUNIverse.
                           addSensor((String)list.get(i), type);
14                     }
15                  :}
16                ;
17 senslist ::=     SENSID:id
18                  {: RESULT = new ArrayList();
19                     RESULT.add(id);
20                  :}
21                | SENSID:id KOMMA senslist:list
22                  {: list.add(id);
23                     RESULT = list;
24                  :}
25                ;
26 senstype ::=     TGSENSORS:type
27                  {: RESULT = type;
28                  :}
29                ;
30 coordblockopt ::=coordblock
31                | /* empty */
32                ;
```

```
33 coordblock ::=   COORDINATES GKA GKZ
34                | COORDINATES GKA coordlist GKZ
35                ;
36 coordlist ::=    coorddef
37                | coorddef coordlist
38                ;
39 coorddef ::=     coordtype:id DOPPELPUNKT coord:coords
                                                SEMIKOLON
40                  {: BestTramNetworkBuilderOfTheUNIverse.
                                    setCoords(id, coords);
41                  :}
42                ;
43 coordtype ::=    SENSID:id
44                  {: RESULT = id;
45                  :}
46                ;
47 coord ::=        KA COORD:x KOMMA COORD:y KOMMA COORD:z KZ
48                  {: RESULT = new int[3];
49                     RESULT[0] = x.intValue();
50                     RESULT[1] = y.intValue();
51                     RESULT[2] = z.intValue();
52                  :}
53                ;
```

Viele dieser Produktionen ist kein Quellcode angehängt. Diese Zeilen sind dennoch für die Erkennung der Grammatik der Datei unverzichtbar – nur findet an diesen Stellen eben keine Weiterverarbeitung von Informationen statt.
Die Zeilen 11-29 zeigen eine Struktur, welche an vielen Stellen im gesamten Parser zu finden ist. Zum einen wird ein Typ erkannt und als String zurückgegeben (Zeile 26-28) und zum anderen wird eine Liste erstellt und gefüllt mit einer Reihe von Objekten – in diesem Fall sind es IDs von Sensoren (Zeile 17-25). Sobald die Liste komplett ist, trifft sie an anderer Stelle mit dem Typ zusammen – an dieser Stelle wird sie dann auch verarbeitet (Zeile 11-16). Hier wird die Liste wieder zerteilt und für jeden Sensor eine Methode im BTNB aufgerufen.
Die Zeilen 30-32 zeigen, wie man eine Optionalisierung eines Blockes einfügt. In den Zeilen (39-42) treffen wieder die Informationen zusammen und werden an den BTNB weiter geschickt. Anhand der Beispieldatei auf Seite 373 lässt sich erkennen, dass die hier erkannte ID die gleiche sein wird, wie im DEFBLOCK – insofern muss bei der Weiterverarbeitung im BTNB erkannt werden, dass es dasselbe Objekt ist.

7.8.5.1.4 BestTramNetworkBuilderOfTheUNIverse Im letzten Schritt werden im BTNB die erhaltenen Informationen in die Klassenstruktur des Simulators (der Klasse „Simulator“) eingefügt. Mit der Methode `getHardware` kann der BTNB auf eine *HashMap* des Simulators zugreifen, indem sämtliche Hardware gesammelt wird.

```
01 protected static void addSensor(String name, String type)
02 {
03     if (type.equalsIgnoreCase("tg-sensors")) {
04         getHardware().put(name, new ToggleSensor(name));
05     }
06 }
07 protected static void setCoords(String name, int[] coords)
08 {
09     Hardware hw = (Hardware) getHardware().get(name);
10     hw.setCoords(coords);
11 }
```

Die erste Methode `addSensor` macht nichts anderes, als ein Objekt vom entsprechenden Typ mit der übergebenen ID zu erzeugen und dieses der HashMap des Simulators hinzuzufügen. Die zweite Methode `setCoords` ist für das Setzen der Koordinaten von bereits existenten Objekten zuständig. Zuerst wird die Hardware über die ID (die als Key der HashMap dient) geholt und für diese dann die Koordinaten gesetzt.
Im BTNB gibt es sehr viele Stellen, an denen gecastet wird oder an denen überprüft werden muss, ob die benötigten Objekte überhaupt existieren. Dies ist der Einfachheit halber für dieses Beispiel an dieser Stelle weggelassen worden.

7.8.5.1.5 Fazit Wichtig für das Projekt TRACS ist an dieser Stelle, dass sowohl der Compiler (der die TND-Datei für den Steuerinterpreter weiterverarbeitet) als auch der Compiler der Simulatorgruppe die gleiche Grammatik einlesen und diese auch auf dieselbe Weise interpretieren. Damit dies möglich ist, muss die Beschreibung der einzulesenden Sprache – der TND – so exakt wie möglich gehalten sein und keinen Interpretationsspielraum bieten.

7.8.5.2 Simulation

Zu der Simulation werden die Implementierungen der wichtigsten und interessantesten Algorithmen vorgestellt. An einigen Stellen wurde eine ganz bestimmte Art der Implementierung gewählt, die erläutert und begründet wird. Die interessanten Stellen der Simulation sind unseres Ermessens nach die Positionsbestimmung der Straßenbahn (d. h. Berechnung einer neuen Position), die Vorausschau auf die nächsten Signale, die sich in Sichtweite befinden, die Vermeidung von Auffahrunfällen und die Kollisionsüberprüfungen, die auf dem physikalischen Weltmodell basieren.

7.8.5.2.1 Positionsbestimmung der Tram Auf Grundlage der Programmstruktur des Bewegungsalgorithmus' 7.8.4.4.2 auf Seite 343 wird nun der Quellcode erklärt, wie die Straßenbahn einen Schritt vorwärts gesetzt wird, bzw. wie die neue Position errechnet wird.

```
01  private Linkable calculatePosition(int[] posArr,
          TrackElement[] teArr, boolean[] dirArr, boolean head) {
02      if ((emerge > 0) && (!head)) {
03          return null;
04      } else {
05          int pos = posArr[0];
06          boolean dir = dirArr[0];
07          TrackElement te = teArr[0];
08          Linkable l = null;
```

Die Methode „calculatePosition“ wird aufgerufen mit vier Parametern. `posArr` enthält die Position der Tram auf dem aktuellen Gleisstück, `teArr` ist das aktuelle TrackElement und `dirArr` bezeichnet die Richtung, in der die Tram auf dem Gleisstück unterwegs ist. Statt Integern und TrackElement werden Arrays diesen Typs übergeben, die nur den einen Integer, bzw. das eine TrackElement enthalten. Der Grund dafür ist, dass die übergebenen Parameter von der Methode verändert werden sollen. Da es in Java™ nicht möglich ist, eine Referenz auf einen primitiven Datentypen wie Integer als Parameter zu übergeben, muss man eine andere Lösung finden. Die Möglichkeit, die hier verwendet wurde, ist, den Integer in einem Array zu verpacken – denn Arrays sind Objekte und Objekte werden immer per Referenz übergeben und können somit von einer Funktion geändert werden (vgl. [Krü00, S. 197]). Letzter Parameter ist ein boolscher Wert, der angibt, ob die Berechung gerade für den Kopf oder das Ende der Bahn stattfindet. Dies ist nötig, um in Situationen wie dem Einfahren oder Verlassen des Gleisnetzes richtige Berechnungen vorzunehmen (beim Einfahren darf sich, solange die Bahn nicht in voller Länge im Netz ist, nur der Kopf voran bewegen, beim Verlassen nur das Ende). Den ersten Einsatz dieses Parameters kann man gleich sehen: In der Variablen `emerge` ist abgelegt, wie viele Schritte die Bahn noch braucht, bis sie in voller Länge im Gleisnetz ist - so lange darf nur der Kopf bewegt werden und entsprechend liefert die Methode `null`, falls in dieser Situation eine neue Position des Endes berechnet werden soll. Der Rückgabewert der Funktion bildet ein Objekt der Klasse `Linkable` und enthält die überfahrenen Hardware-Elemente, falls in der Berechnung der Fortbewegung festgestellt wird, dass ein TrackElement verlassen und ein neues betreten wurde. Wurde keine Hardware überfahren, so liefert die Methode `null` zurück.

```
09          pos = pos + (dir ? 1 : -1);
10          if (((pos > te.getLength()-1) || (pos < 0)) &&
               ((!head) || (!headVanish))) {
{
```

Dann wird die Position abhängig von der Richtung, in die die Tram unterwegs ist, um Eins verändert. Ein TrackElement hat eine feste Länge und die Tram kann sich nur an den Stellen 0-(Länge-1) befinden (s. Skizze 7.80 (in diesem Beispiel Länge 5)). Wenn die neue Position größer als (Länge-1) oder kleiner als 0 ist, so wurde das TrackElement verlassen und die neue Position und Richtung auf dem neuen TrackElement müssen berechnet werden. Falls jedoch der Kopf das Ende des Gleisnetzes erreicht hat, darf für ihn kein neues TrackElement berechnet werden, da sich in dieser Situation nur noch das Ende der Bahn weiterbewegt (abgesehen davon gibt es am Ende des Gleisnetzes kein neues TrackElement mehr). In `headVanish` ist abgelegt, ob der Kopf an einem Endpunkt angekommen ist - weitergerechnet werden darf nur, wenn entweder nicht für den Kopf gerechnet wird oder er nicht am Ende angekommen ist.

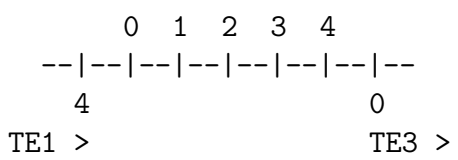


Abbildung 7.80: Implementierung: Skizze eines TrackElements

```
11          if (!(te.getBorders()[(dir ? 1 : 0)]
                  instanceof Linkable)) {
12             failure("Tram moves on unknown hardware element "+
13                   "(no Linkable)");
            }
```

Abhängig von der Richtung, auf der die Tram auf dem TrackElement unterwegs ist, wird das entsprechende Ende des TrackElements geholt. Am Ende eines TrackElements befindet sich ein Hardware-Element, dass das Interface `Linkable` implementiert – andernfalls befände sich dort ein Element, welches an dieser Stelle nicht stehen darf und bei der Initialisierung wurde ein Fehler gemacht. In diesem Fall muss die Simulation gestoppt werden (Methode `failure()`).

```
14          } else {
```

```
15              l = (Linkable) te.getBorders()[(dir ? 1 : 0)];
16              try {
17                  te = l.getNextTE(te, head, false);
18                  if (l == te.getBorders()[0]) {
19                      dir = true;
20                  } else {
21                      dir = false;
22                  }
23                  if (dir == true) {
24                      pos = 1;
25                  } else {
26                      pos = te.getLength()-2;
27                 }
28              } catch (NetworkEndException ne) {
29                  if (((l instanceof Sensor) &&
30                          ((Sensor)l).getIsDeadEnd())
31                      || ((l instanceof Mark) &&
32                          ((Mark)l).getIsDeadEnd())) {
33                              Simulator.getInstance().
34                                  handleFailureSI
35                                  ("Tram directed to dead end");
36                  } else {
37                      if (this.headVanish) {
38                          this.vanish = true;
39                      } else {
40                          this.headVanish = true;
41                      }
42                  }
43              }
44          }
45      }
```

Wenn die Prüfung des Hardware-Elements erfolgreich war, so folgt die Berechnung des neuen TrackElements, der neuen Richtung und der neuen Position. Zunächst wird das Hardware-Element, auf dem sich die Tram nun befindet, in `Linkable` gecastet (dass dies ohne Fehlermöglichkeit geht, ist durch Zeile `11` garantiert). Ein Objekt der Klasse `Linkable` hat die Methode `getNextTE()`, die das nächste TrackElement zurück gibt und als Parameter das alte TrackElement erwartet.
Dann wird in Zeile `18` geprüft, ob die Linkable-Hardware, die die Tram gerade überfährt, bei dem TrackElement als Border[0] oder Border[1] eingetragen ist. Die Richtung eines TrackElements ist anhand seiner beiden Border definiert: die Bewegung von Border[0]

nach Border[1] ist als positiv definiert, die andere als negativ (s. Skizze 7.80 auf Seite 379 (Richtung als `>` dargestellt)). Entsprechend des Rückgabewerts von `te.getBorders()` wird die neue Richtung auf `true` oder `false` gesetzt (Zeilen `19` und `21`).
Abhängig von der neuen Richtung wird dann die neue Position auf dem TrackElement entschieden. Bewegt sich die Tram auf dem neuen TrackElement positiv, so setzt sie bei dem TrackElement an Stelle 1 an; andernfalls an Stelle (Länge-2) (Zeilen `23-27`).
Die Berechnung der neuen Position befindet sich in einem try-catch-Block, der dann zum Tragen kommt, wenn die Bahn am Ende des Gleisnetzes angekommen sein sollte. Dort gibt es kein neues TrackElement mehr, `getNextTE()` wirft in diesem Fall eine `NetworkEndException`. Tritt eine solche auf, wird zunächst geprüft, ob das nun erreichte Ende ein totes Ende ist – dorthin dürfte eine Bahn nicht geleitet werden, weswegen in dem Fall eine Fehlersituation eingetreten ist. Handelt es sich aber um eine Ausfahrt, so wird geprüft, ob der Kopf bereits das Netz verlassen hat – wenn ja, so muss nun auch das Ende der Bahn das Netz verlassen, womit sie völlig das Netz verlassen hätte (das wird in `vanish` gespeichert). Andernfalls ist es erst der Kopf, welcher verschwindet, so dass die zugehörige Variable gesetzt wird. (Zur Erinnerung: Wenn der Kopf verschwunden ist, die Berechnung aber für den Kopf vorgenommen werden soll, gelangt der Programmlauf nicht an diese Stelle.)
Damit ist die Berechnung der neuen Position auf dem neuen TrackElement abgeschlossen.

```
46      if ((!head) || (!headVanish)) {
47          dirArr[0] = dir;
48          posArr[0] = pos;
49          teArr[0]  = te;
50      }
51      if ((this.justEmerged) && (!head)) {
52          l = (Linkable) te.getBorders()[(dir ? 0 : 1)];
53          this.justEmerged = false;
54      }
55      return l;
56    }
57  }
```

Zum Schluss werden die errechneten Werte in die übergebenen Arrays geschrieben und das überfahrene Linkable-Hardware-Element zurückgegeben. Außerdem erfolgt an dieser Stelle noch eine Abfrage, ob eine Bahn in gerade diesem Schritt vollständig in das Netz eingefahren ist, in dem Fall muss nämlich das an der Einfahrt befindliche Element (ggf. ein Sensor, der reagieren muss) ermittelt werden.

7.8.5.2.2 Signalvorausschau der Tram Signale können an allen Linkable-Hardware-Elementen vorkommen. Da es möglich ist, dass mehrere solche Elemente in kurzem Abstand aufeinander folgen, muss die Tram in der Lage sein, bis hin zu einer bestimmten Sichtweite (die Weite, aus der sie garantiert noch zum Halt kommen kann, ohne ein Signal zu überfahren) voraus zuschauen – auch über mehrere Linkable-Hardware-Elemente hinweg. Dabei muss der Zustand aller innerhalb dieser Sichtweite befindlichen Signale gespeichert und bei Erreichen des jeweiligen Signals entsprechend angehalten (falls Stop-Signal früh genug gesehen) oder durchgefahren (falls Stop zu spät gesehen oder Signal auf Fahrt) werden.
Dazu wird der folgende rekursive Algorithmus benutzt:

```
01  private boolean checkStop(TrackElement te, int pos,
02      boolean dir, int steps, int iter) {
```

Übergabeparameter sind aktuelle Kopf-Position der Bahn (TrackElement, Position darauf, Fahrtrichtung), außerdem die noch vorauszuschauende Sichtweite (`steps`) und die Iterationstiefe `iter`, angegeben als Zweierpotenz (z.B. Rekursionstiefe 0: iter=1, Rekursionstiefe 2: iter=4 etc. s. u.).

```
03      boolean stop = false;
04      Hardware hw = te.getBorders()[dir ? 1 : 0];
05      int length = te.getLength();
06      if (hw instanceof Linkable) {
07          Linkable l = (Linkable) hw;
08          boolean sigExists = l.hasSignal(te);
```

Anhand der Position und Fahrtrichtung der Tram wird das nächste Linkable-Hardware-Element gesucht und geprüft, ob dort ein Signal steht, das für die Richtung der Bahn zuständig ist.

```
09          if (sigExists) {
10              int state = l.getSignal(te).getState();
11              if ((state == Signal.POS_STOP) ||
12                (!route.isSignalPositionCorrect(
13                    l.getSignal(te), state))) {
14                  if ((dir ? (pos <= length - 1 - steps) :
15                    (pos >= steps)) || (!getIsMoving())) {
16                      if (((mustStop / iter) % 2 ) == 0) {
17                          logger.info("Tram " + id +
18                            ": Wrong signal percepted in" + iter
19                            + ". iteration");
20                          mustStop += iter;
21                      }
22                  }
```

Ist dem so, wird dessen Zustand geprüft – steht es auf Stop oder in einer Stellung, die zwar Fahrt erlaubt, aber nicht der für die Route der Bahn nötigen Richtung entspricht (das wird mittels `route.isSignalPositionCorrect()` geprüft) und ist die Tram entweder noch weit genug vom Signal entfernt (d.h. das Signal ist weiter entfernt als die minimale Vorausschau-Weite) oder steht die Tram gerade (dann kann sie ein Signal in jedem Fall beachten), so wird der Stop-Zustand als erkannt gespeichert. Dies geschieht in der Integer-Variablen `mustStop`. Diese ist als Bitmap codiert – das nullte Bit steht hierbei für das nächste Linkable-Hardware-Element, das erste für das darauf folgende usw. Befindet sich am jeweiligen Linkable-Element ein Signal, das auf Stop steht, so wird das entsprechende Bit gesetzt (`mustStop += iter` – daher auch die besondere Darstellung der Iterationstiefe, so kann man ihren Wert direkt benutzen, um `mustStop` zu manipulieren). Das Setzen geschieht natürlich nur, falls das Bit noch nicht gesetzt war.

```
23              } else {
24                  logger.info("Tram " + id +
25                      ": Correct signal percepted in" +
26                      iter + ". iteration"););
27                  if (((mustStop / iter) % 2 ) == 1) {
28                      mustStop -= iter;
29                  }
30              }
```

Ist zwar ein Signal vorhanden, steht es aber nicht auf Stop und nicht in einer für die gewünschte Route falschen Position, so wird ein eventuell gesetztes Bit in `mustStop` zurückgenommen. Das geschieht unabhängig von der Entfernung zum Signal, da ja auch die Situation denkbar ist, dass das Signal kurz bevor die Bahn es erreicht, die Fahrt freigibt. Regel: Stop kann nur dann beachtet werden, wenn man es rechtzeitig vorher sieht – Fahrtfreigabe kann immer beachtet werden.

```
31          } else {
32              if (((mustStop / iter) % 2 ) == 1) {
33                  mustStop -= iter;
34              }
35          }
```

Ist kein Signal vorhanden, wird selbstverständlich das Bit nicht gesetzt – die eventuelle Rücknahme eines gesetzten Bits ist nur pro forma.

```
36          int dist = (dir ? (length - 1 - pos) : (pos));
37          if (steps > dist) {
38              try {
39                  te = l.getNextTE(te, false, true);
```

```
40                  if (l == te.getBorders()[0]) {
41                      dir = true;
42                  } else {
43                      dir = false;
44                  }
45                  if (dir == true) {
46                     pos = 0;
47                  } else {
48                      pos = te.getLength()-1;
49                  }
50                  checkStop(te, pos, dir, steps-dist, iter*2);
51              } catch (NetworkEndException ne) {
52                  logger.info("Previewing network end or " +
53                   " turnout in undefined position");
54              }
55          }
```

Hier erfolgt nun die weitere Vorausschau: Befindet sich die Bahn in einer geringeren Entfernung zum nächsten Linkable-Element, als sie vorausschauen muss, so ist es erforderlich, darüber hinauszuschauen. Es wird dann eine neue Position der Bahn berechnet, und zwar diejenige, die sie einnehmen würde, wenn sie gerade das nächste Linkable-Element erreicht hätte, also Kopf direkt darauf. Der letzte Parameter `true` in `getNextTE` bewirkt hierbei, dass die Positionsberechnung nur im Voraus erfolgt und nicht wie beim tatsächlichen Weiterfahren Wirkungen auf die überfahrenen Hardware-Elemente ausgeübt werden (Sensorauslösungen, Weichen passiv schalten). Anschließend wird die Methode mit dieser neuen Kopfposition, einer entsprechend verringerten Vorausschau-Weite und der nächsten Iterationstiefe aufgerufen. Wenn die Vorausschau bis zum Ende des Netzwerkes reicht, muss natürlich nicht weiter gesehen werden.

```
56      } else {
57          failure("Unknown type of hardware in preview!");
58      }
59
60      if ((getMustStop() % 2 == 1) &&
61              (getHeadDir() ?
62              (getHeadPos() == getHeadElement().getLength()
                    - 2) :
63              (getHeadPos() == 1))) {
64          logger.info("Tram " +getId()+ " must stop in front of "
65              + "signal while " + missingSteps + " steps " +
                + "are pending");
66          stop = true;
```

```
67          missingSteps = 0;
68      }
69      return stop;
70  }
```

Ist die Vorausschau gemäß der gewünschten Weite abgeschlossen (diese Weite ist in einer Konstanten der `Tram`-Klasse abgelegt), gelangt der Programmablauf an diese Stelle: es wird nun geprüft, ob am aktuell nächsten Linkable-Element ein Stop-Signal zu beachten ist, indem der in `mustStop` abgelegte Wert abgefragt wird. Falls dem so ist und die Bahn an der Position ist, an der sie stehen bleiben muss, liefert die Methode true zurück, so dass die Methode, die „checkStop“ aufgerufen hat, die Bahn anhalten kann. Zu beachten ist noch, dass für die korrekte Funktion jedes Mal, wenn die Bahn ein Linkable-Element überfährt, `mustStop` per Ganzzahldivision durch 2 dividiert wird, da ja die in der Bitmap gespeicherten Werte, die wie beschrieben ein Signal am nächsten, übernächsten usw. Linkable-Element referenzieren, entsprechend verschoben werden müssen.

7.8.5.2.3 Auffahren & Kollisionsüberprüfung der Tram Prinzipiell gibt es drei Arten der Kollision:

- eine Bahn fährt auf das Ende einer vorausfahrenden Bahn auf
- Bahnen stoßen mit dem Kopf frontal zusammen
- eine Bahn fährt einer anderen in die Seite

Der erste Fall soll nicht durch den SI, sondern durch den Fahrer verhindert werden. Insofern muss der Simulator, bevor er die Bahn tatsächlich fortbewegt, bestimmen, ob dieser Fall eintreten würde und die Bahn nur einen Schritt weitersetzen, wenn dieser Fall nicht vorliegt. Zur Erinnerung: die Position einer Bahn bestimmt sich durch vier Angaben:

- TrackElement, auf dem sich der Kopf befindet
- genaue Position des Kopfes auf diesem TrackElement
- TrackElement, auf dem sich das Ende befindet
- genaue Position des Endes auf diesem TrackElement

Die Algorithmen zur Überprüfung, ob eine Tram auf das Ende einer anderen Tram auffährt und die Kollisionsüberprüfung, ob es zu einem Frontalzusammenstoß gekommen ist, sind sehr ähnlich. Der eine prüft, ob die Kopfposition einer Bahn mit der Endposition einer anderen Bahn identisch ist, der andere prüft, ob zwei Köpfe die gleiche Position haben. Aus diesem Grund wird an dieser Stelle nur der Quellcode von

„driveInto()“ (Auffahren) erklärt und die Stellen, an denen der Algorithmus zu der Frontal-Kollisionsprüfung abweicht, entsprechend angesprochen.
Die Kollisionsüberprüfung, ob eine Tram einer anderen in die Seite hinein gefahren ist, wird nicht anhand von Quellcode erläutert, da es sich um eine sehr simple Methode handelt (s. Konzept 7.8.4.4.2 auf Seite 351). Es wird lediglich abgefragt, ob ein Hardware-Element, auf das eine Bahn in der aktuellen Runde gefahren ist, schon von einer anderen Bahn besetzt ist oder in eben dieser Runde schon von einer anderen Tram besetzt war. (Die andere Bahn könnte vor der zu prüfenden Bahn in dieser Runde bereits von dem HW-Element fortbewegt worden sein, ein Zusammenstoß ist es aber dennoch, da die Bewegungen aller Bahnen in einer Runde ja gleichzeitig sein sollen.)
Nun der Quellcode von „driveInto()“:

```
01  private boolean driveInto(ArrayList trams,
        TrackElement newHElement, int newHPos) {
```

Die Methode erwartet als Parameter `trams` (eine Liste aller Trams im Gleisnetz), das neue TrackElement, das der Kopf der Tram befährt (`newHElement`), und die Position auf dem neuen Gleisstück (`newHPos`). Der Rückgabewert sagt aus, ob die Tram auf eine andere Tram auffahren würde.
„checkHeadCollision()“ erwartet nur die Liste aller Straßenbahnen als Parameter. Das TrackElement und die Position sind die aktuellen Werte der Tram. Denn: „driveInto()“ wird *vor* dem Bewegen der Tram aufgerufen (und die Tram dann nicht bewegt, falls ein Auffahren stattfinden würde) und „checkHeadCollision()“ *danach*.

```
02    Tram tram;
03    for (int i = 0; i < trams.size(); i++)  {
04      tram = (Tram) trams.get(i);
05      if (!(trams[i].equals(this))) {
```

Als Erstes wird die Liste aller Trams durchgegangen und bei jedem Eintrag wird geprüft, ob es sich nicht um die Tram selbst handelt.

```
04        if ((tram.getTailElement() == newHElement) &&
05            (tram.getTailPos() == newHPos)) {
06                  logger.info("Tram " + id +
                    " must stop to prevent head-to-end collision "
                        + "while " + missingSteps + " pending");
07          this.isMoving = false;
08          missingSteps = 0;
09          return true;
10        }
11      }
12    }
```

Wenn in der Liste eine andere Tram gefunden wurde, so wird überprüft, ob ihr Ende das gleiche TrackElement und die gleiche Position hat, auf die die Tram hinbewegt werden soll. Falls Ja, so wird im Logger vermerkt, dass ein Auffahren erkannt wurde und die Tram wird gestoppt. Die verbleibenden Schritte, die gefahren werden sollen, werden auf 0 gesetzt und die Methode wird mit dem Wert `true` verlassen.
Bei der Kollisionsüberprüfung werden zwei andere Positionen verglichen: Befindet sich der Kopf einer anderen Tram auf der gleichen Position auf dem gleichen TrackElement wie die Tram selbst, so ist es zu einem Zusammenstoß gekommen. Wenn dies der Fall ist, so wird das im Logger entsprechend vermerkt, im Simulator wird eine handleCrash-Methode aufgerufen und die Tram merkt sich selbst, dass sie mit einer anderen Tram zusammengestoßen ist.

```
13    return false;
14  }
```

Wurde kein Zusammenstoß festgestellt, so wird die Methode mit dem Wert `false` verlassen.

7.8.5.3 Graphical User Interface

Bei dem Graphical User Interface (GUI) sind alle Methoden recht simpel gehalten. Der Algorithmus, wie die Gleisnetz-Koordinaten in die Fenster-Koordinaten umgerechnet werden, wurde bereits ausführlich in 7.8.4.5.1 auf Seite 359 erklärt und die Implementierung in Java™ist nicht von Belang. Auch die Methoden zum Zeichnen des Gleisnetzelemente sind nicht kompliziert, da es im Wesentlichen um Zeichnen von Kreisen und Linien an bestimmten Koordinaten geht, wobei diese Elemente grafisch relativ simpel gehalten sind. Am interessantesten erscheint das Zeichnen der Tram, da es hier mit dem Fahren über Hardware-Elemente (so dass sich die Bahn also über mehrere TrackElemente erstreckt) eine gewisse Besonderheit gibt.
Grundsätzlich existieren zwei Möglichkeiten: Die Tram fährt vollständig auf *einem* Gleisstück oder Kopf und Ende befinden sich auf zwei *unterschiedlichen* TrackElementen.

7.8.5.3.1 Zeichnen eines Bahn-Abschnittes Als Bahn-Abschnitt wird in diesem Zusammenhang derjenige Teil einer Bahn verstanden, der sich auf einem TrackElement befindet. Das kann also von einem sehr kurzen Abschnitt bis hin zur vollständigen Bahn alles sein.
Es wurde eine ganz spezielle Implementierung gewählt, um aufwendige Rechnungen beim Zeichnen zu ersparen. Der Zeichenmethode werden die Koordinaten des Kopfes und des Endes eines Straßenbahnabschnittes gegeben. In Abbildung 7.82 auf Seite 390 sind diese beiden Stellen durch ein `x` gekennzeichnet. Statt auf dieser Grundlage umständlich die vier Ecken des roten Rechtecks zu berechnen und den Mittelpunkt des Kreises zu

berechnen, wird nur der Punkt auf der Geraden zwischen Kopf und Ende berechnet, bis zu dem das dunkle Rechteck – der Körper der Tram – reicht und der den Mittelpunkt des Kreises (Kopfes) bildet (Stelle 2 in der Skizze).
Für das Zeichnen eines Abschnittes wurde folgende Hilfsmethode implementiert:

```
public void drawTram(Graphics g, int type, int xHead,
        int yHead, int xTail, int yTail, int width) {
```

Die Methode erwartet als Parameter den Grafikkontext `g`, die x- und y-Koordinaten des Anfangs dieses Abschnittes in Fensterkoordinaten (`xHead`, `yHead`), die beiden Koordinaten des Endes dieses Abschnittes (`xTail`, `yTail`, in Fensterkoordinaten) und `width` – die Breite der Tram im Fenster. Interessant ist auch der Parameter `type` – dieser gibt an, ob es sich um denjenigen Abschnitt der Bahn handelt, der den Kopf beinhaltet, oder einen anderen Abschnitt. Die Methode liefert nichts zurück, sondern zeichnet direkt im Fenster.

```
Graphics2D g2 = (Graphics2D) g;
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                    RenderingHints.VALUE_ANTIALIAS_ON);
```

Als erstes wird der Grafikkontext in eine 2D-Umgebung gecastet, um mehr Kontrolle, besonders über Geometrie, zu erlangen (s. [Sun, Klasse „Graphics2D“]). Des Weiteren wird Antialiasing eingeschaltet.

```
double xDiff = xHead - xTail;
double yDiff = yHead - yTail;
double radius = width/2;
double abstand = (int) Math.sqrt(xDiff*xDiff +
    yDiff*yDiff);
double xNeu = xTail;
double yNeu = yTail;
if (abstand > 0) {
    xNeu += xDiff * ((abstand-radius)/abstand);
    yNeu += yDiff * ((abstand-radius)/abstand);
}
```

Hier wird mathematisch der Punkt 2 aus Abbildung 7.82 auf Seite 390 bestimmt. Sowohl x- als auch y-Koordinate müssen berechnet werden. Anschaulich sieht die Formel zur

Berechnung des Punktes wie folgt aus, wobei *Radius* den Radius des Kreises bezeichnet:

$$Abstand = \sqrt{(x2-x1)^2+(y2-y1)^2} \tag{7.8}$$

$$\frac{xNeu}{yNeu} = \frac{x1}{y1} + \frac{x2-x1}{y2-y1} * \frac{Abstand - Radius}{Abstand} \tag{7.9}$$

xNeu und *yNeu* sind die Koordinaten des Punktes, der in der Skizze auf Höhe von `2` liegt.

```
g2.setStroke(new BasicStroke(width,
             BasicStroke.CAP_BUTT,
             BasicStroke.JOIN_MITER));
g2.setColor(Color.red);
g2.drawLine(xTail, yTail, (int)xNeu, (int)yNeu);
```

Mittels `setStroke` wird die Dicke der Linie gesetzt, sozusagen die Pinselbreite mit der gemalt werden soll. Diese Methode ist eine exklusive `Graphics2D`-Methode. Der Parameter `width` in Zeile `10` gibt die Dicke an, `BasicStroke.CAP_BUTT` bedeutet, dass der Pinselstrich keine abgerundeten Kanten haben soll und auch nicht über die Enden hinweg ragen soll (dies ist mittels eines anderen Parameters möglich). Der letzte Parameter hat für das Zeichnen von Linien keine Bedeutung.
Anschließend wird die Pinselfarbe auf Rot gesetzt und vom Ende der Tram bis zu dem neu errechneten Punkt eine dicke Linie gezogen. Damit ist der Rumpf der Straßenbahn gezeichnet.

```
        if (type == HEAD_TAIL) {
            g2.setStroke(new BasicStroke(width,
                     BasicStroke.CAP_ROUND,
                     BasicStroke.JOIN_MITER));
            g2.setColor(Color.cyan);
            g2.drawLine((int)xNeu, (int)yNeu,
                     (int)xNeu, (int)yNeu);
        }
    }
```

Falls es sich bei dem Abschnitt um denjenigen mit dem Kopf handelt, wird nun der Kopf gezeichnet. Es wird weiterhin die Methode `setStroke` verwendet. Wieder wird die Breite der Tram als Parameter übergeben und der zweite Parameter lautet diesmal `BasicStroke.CAP_ROUND`, was bedeutet, dass Ecken abgerundet sein sollen. Die Ecken werden so stark abgerundet, dass sich ein Halbkreis am Ende der Linie bildet. Es wird

jedoch nichts von der Linie abgeschnitten, sondern die Linie wird um `width`/2 verlängert (s. Abbildung 7.81).
Dann wird die Farbe auf Cyan gesetzt und eine Linie gezeichnet, die nur aus einem Punkt besteht. Durch `BasicStroke.CAP_ROUND` wird die Linie aber verlängert und abgerundet, so dass sich ein Kreis ergibt.

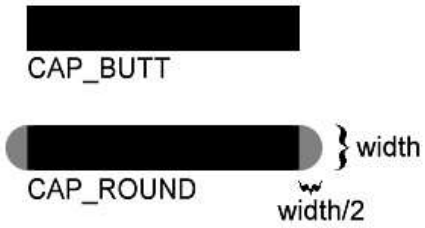


Abbildung 7.81: Tram zeichnen: Unterschiede bei Pinselstrichen

Abbildung 7.82: Tram zeichnen: Koordinaten

7.8.5.3.2 Zeichnen der gesamten Bahn Die gesamte Bahn besteht je anch ihrer Position aus einem bis hin zu vielen Abschnitten. Darunter ist jedoch nur ein Abschnitt, der den Kopf enthält – alle anderen sehen, abgesehen von ihrer Länge, gleich aus. Dies wird ausgenutzt, indem die Abschnitte der Bahn von hinten nach vorne durchgegangen werden, der letzte zu zeichnende Abschnitt ist dann immer derjenige mit dem Kopf.

```
private void drawTram(Graphics g, Tram tram) {
    int[] headCoords = new int[2];
    TrackElement headTE = tram.getHeadElement();
    int headPos = tram.getHeadPos();
    int[] tailCoords = new int[2];
    TrackElement tailTE = tram.getTailElement();
    int tailPos = tram.getTailPos();
    boolean tailDir = tram.getTailDir();
    int[] hwCoords = new int[2];
    TrackElement interTE = null;

    Hardware[] borders;
    int length;
```

```
int[][] coords = new int[2][];

borders = tailTE.getBorders();
length  = tailTE.getLength();
coords[0] = calcCoords(borders[0].getCoords());
coords[1] = calcCoords(borders[1].getCoords());

tailCoords[0] = coords[0][0]
    + (int) (((double)tailPos / (double)(length))
             * (coords[1][0] - coords[0][0]));
tailCoords[1] = coords[0][1]
    + (int) (((double)tailPos / (double)(length))
             * (coords[1][1] - coords[0][1]));
```

Die Position der Bahn im Gleisnetz wird geholt, die Position des Endes dabei bereits in Pixelkoordinaten umgerechnet. Nun beginnt die Unterscheidung:

```
while (headTE != tailTE) {
    if (tailDir) {
        headCoords = coords[1];
    } else {
        headCoords = coords[0];
    }
```

Falls Bahn-Kopf und Bahn-Ende nicht auf dem gleichen TE zu finden sind, bedeutet dies, dass der letzte Abschnitt vom Ende der Bahn bis zu einem Ende des TEs reicht (abhängig von der Fahrtrichtung ist, welches Ende des TE berücksichtigt werden muss).

```
    drawTram(g, TAIL_TAIL, headCoords[0], headCoords[1],
        tailCoords[0], tailCoords[1], TRAM_WIDTH);
    tailCoords[0] = headCoords[0];
    tailCoords[1] = headCoords[1];
```

Der Abschnitt wird gezeichnet. Als nächste Endkoordinaten werden nun die Koordinaten des vordersten Punktes des gezeichneten Abschnittes gesetzt.

```
    if (tailDir) {
      try {
        tailTE = ((Linkable)borders[1]).
          getNextTE(tailTE, false, true);
```

```
            if (borders[1] == tailTE.getBorders()[0]) {
              tailDir = true;
            } else {
              tailDir = false;
            }
          } catch (NetworkEndException ne) {
            break;
          }
        } else {
          try {
            tailTE = ((Linkable)borders[0]).
              getNextTE(tailTE, false, true);
            if (borders[0] == tailTE.getBorders()[0]) {
              tailDir = true;
            } else {
              tailDir = false;
            }
          } catch (NetworkEndException ne) {
            break;
          }
        }
        borders = tailTE.getBorders();
        length  = tailTE.getLength();
        coords[0] = calcCoords(borders[0].getCoords());
        coords[1] = calcCoords(borders[1].getCoords());
    }
```

Es wird nun mittels der Funktionen für die Fortbewegung der Bahn das vom Ende der Bahn aus nächstvordere TrackElement ermittelt und zum Beginn der while-Schleife zurückgekehrt. Befindet sich der Kopf auch nicht auf diesem TE, heißt das, dass der nächste Abschnitt der Bahn sich über das ganze TE erstreckt, dieses wird gezeichnet, das nächste TE weiter vorn geholt, die Position des Kopfes geprüft usw...

```
headCoords[0] = coords[0][0]
    + (int) (((double)headPos / (double)(length))
         * (coords[1][0] - coords[0][0]));
headCoords[1] = coords[0][1]
    + (int) (((double)headPos / (double)(length))
         * (coords[1][1] - coords[0][1]));
drawTram(g, HEAD_TAIL, headCoords[0], headCoords[1],
    tailCoords[0], tailCoords[1], TRAM_WIDTH);
```

```
}
```

...bis man dann tatsächlich das TE erreicht, auf dem der Kopf ist. Nun muss noch der vordere Abschnitt der Bahn gezeichnet werden und die Bahn ist komplett dargestellt.

7.8.5.4 Schnittstelle zu den Treibern

Wie bereits im Konzept erwähnt, wird die Kommunikation zu den Hardware-Treibern und damit indirekt zum Steuerinterpreter via Netzwerk, im speziellen die Internet-Protokollfamilie, realisiert. Für jede der Kommunikationsrichtungen existiert ein eigenes Subsystem, das als eigener Thread läuft. Damit wird sichergestellt, dass die Kommunikation zu jeder Zeit möglich ist und nicht beispielsweise durch einen längeren Rechenvorgang des Simulators blockiert wird.

7.8.5.4.1 Richtung Treiber → Simulator Es folgt nun ein Auszug aus dem Code für die o.a. Kommunikationsrichtung, dabei werden die in der eigentlichen Datei vorhandenen Kommentare ausgelassen, auch ist der Code nicht komplett aufgeführt. Die Betrachtung beschränkt sich auf das für den Betrieb Wesentliche.

```
public class DriverRequestServer extends Thread {
    private LinkedList requestList;
    private LinkedList driverList;
    private final int PORT = 12345;
```

Der Server, der die Anfragen der Treiber annehmen soll, verfügt über zwei Variablen – einerseits eine Liste, in der die eingegangenen Anfragen zwischengespeichert werden, bis der Simulator sie abruft, andererseits eine ganz ähnluch funktionierende Liste, in der Informationen über Treiberinstanzen abgelegt werden, die sich erstmals mit dem Simulator verbunden und ein Init-Request gesendet haben. Außerdem liegt als Konstante die Portnummer vor, auf der dieser Server betrieben wird und an die die Treiber ihre Anforderungen richten müssen.

```
public void run() {
    this.requestList = new LinkedList();
    this.driverList = new LinkedList();
    ArrayList clientSockList = new ArrayList();
    ArrayList readerList = new ArrayList();
    ServerSocket serverSocket = null;

    boolean success = true;
    try {
```

```
        serverSocket = ServerSocketChannel.open();
        serverSocket.socket().bind(new InetSocketAddress(PORT));
        serverSocket.configureBlocking(false);
    } catch (IOException e) {
        System.out.println("Port " + PORT + " not usable!");
        success = false;
    }
```

Die Klasse muss die Methode `run` implementieren, um als Thread laufen zu können. Zunächst werden in dieser Methode die Listen für die Anfragen und Treiberinfos angelegt, außerdem zwei weitere Listen, die einerseits die verschiedenen offenen Sockets, andererseits die dafür existierenden Lesepuffer enthalten. Es wird dann versucht, einen Socket auf dem bereits bezeichneten Port zu öffnen. Ist das nicht möglich, erfolgt eine Meldung darüber, so dass der Nutzer des Simulators an dieser Stelle informiert wird und die nötigen Schritte einleiten kann, um den Start wie gewünscht zu ermöglichen.

```
    while (success) {
        SocketChannel clientSocket = null;
        String request = "";
        try {
            clientSocket = serverSocket.accept();
            if (clientSocket != null) {
                clientSocket.configureBlocking(false);
                clientSockList.add(clientSocket);
                readerList.add(new StringBuffer());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        for(int i=0; clientSockList.size()>i; i++) {
            try {
                SocketChannel s = (SocketChannel)
                  clientSockList.get(i);
                StringBuffer buf = (StringBuffer)
                  readerList.get(i);
                int nlpos;
                while ((nlpos=buf.indexOf("\n")) >= 0) {
                    if (nlpos == 0) {
                        buf.deleteCharAt(0);
                        continue;
```

```
                }
                request = buf.substring(0, nlpos);
                buf.delete(0, nlpos + 1);
                System.out.println("Request received: "
                  + request);
                if (initRequest(request)) {
                    addDriver(request, s);
                }
                if (checkRequest(request)) {
                    addRequest(request);
                }
            }
            try {
                int len;
                ByteBuffer tmpbytebuf =
                  ByteBuffer.allocate(100);
                while((len = s.read(tmpbytebuf)) != 0) {
                    if ((!s.isConnected())||(len < 0)) {
                        s.socket().close();
                        clientSockList.remove(i);
                        readerList.remove(i);
                        i--;
                        System.out.println("Socket closed");
                        break;
                    }
                    byte tmpbuf[] = new byte[len];
                    tmpbytebuf.rewind();
                    tmpbytebuf.get(tmpbuf);
                    String tmpstr = new String(tmpbuf,
                      "US-ASCII");
                    buf.append(tmpstr);
                    tmpbytebuf = ByteBuffer.allocate(100);
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    try {
```

```
                Thread.sleep(100);
            } catch (Exception e) {
            }
        }
    }
```

Anschließend läuft in einer Endlosschleife der folgende Vorgang:
Es wird geprüft, ob eine Anfrage durch einen externen Prozess am Server-Port vorliegt. Sobald diese erfolgt, wird die Verbindung angenommen und der entstandene ClientSocket (in Form eines SocketChannel) der Liste der ClientSockets hinzugefügt. Ebenfalls wird ein Lesepuffer für ihn angelegt. Anschließend wird über alle vorliegenden ClientSockets iteriert: Befindet sich im zugehörigen Lesepuffer ein Zeilenende-Zeichen, so wird davon ausgegangen, dass es sich um eine komplette Meldung des angeschlossenen Treibers handelt. Diese wird aus dem Puffer entfernt und nachfolgend auf ihren Typ überprüft. Ist es ein Init-Request, werden die Daten über den neuen Treiber (für welches HW-Element ist er zuständig, unter welchem Port wartet er auf Meldungen) in der dafür vorgesehenen Liste abgelegt. Ist es dagegen eine Anforderung auf Zustandsänderung des Elements, wird der String mittels `addRequest` der Liste der Anfragen hinzugefügt. Entspricht der String keinem der beiden vorgesehenen Formate, wird er verworfen. Als letzter Schritt in der Iteration über die Sockets wird nun versucht, etwas aus dem Socket zu lesen und dem zugehörigen Puffer hinzuzufügen. Sollte der Socket inzwischen geschlossen sein, werden die Einträge in der Liste der Sockets und der Puffer entfernt. Anschließend wird der nächste Socket überprüft, und wenn alle Sockets durchlaufen wurden, beginnt die Endlosschleife von neuem.

```
public boolean checkRequest(String request) {
    String actToken;
    StringTokenizer st = new StringTokenizer(request);
    if (st.countTokens() != 3) {
        return false;
    } else {
        actToken = st.nextToken();
        if (!actToken.equals("request")) {
            return false;
        } else {
            actToken = st.nextToken();
            switch (actToken.charAt(0)) {
                case '1':
                    if (actToken.length() != 4) {
                        return false;
```

```
        } else {
            actToken = st.nextToken();
            if ((actToken.equals("quit")) ||
              (actToken.equals("left")) ||
              (actToken.equals("straight")) ||
              (actToken.equals("bent")) ||
              (actToken.equals("right"))) {
                return true;
            } else {
                return false;
            }
        }
    case '2':
        if (actToken.length() != 4) {
            return false;
        } else {
            actToken = st.nextToken();
            if ((actToken.equals("quit")) ||
              (actToken.equals("straight")) ||
              (actToken.equals("bent"))) {
                return true;
            } else {
                return false;
            }
        }
    case '3':
        if (actToken.length() != 4) {
            return false;
        } else {
            actToken = st.nextToken();
            if ((actToken.equals("quit")) ||
              (actToken.equals("left")) ||
              (actToken.equals("right")) ||
              (actToken.equals("straight")) ||
              (actToken.equals("stop")) ||
              (actToken.equals("waitright")) ||
              (actToken.equals("waitstraight")) ||
              (actToken.equals("waitleft")) ||
              (actToken.equals("rrrighton")) ||
              (actToken.equals("rrlefton")) ||
              (actToken.equals("rrstraighton")) ||
```

```
                                (actToken.equals("rrrightoff")) ||
                                (actToken.equals("rrleftoff")) ||
                                (actToken.equals("rrstraightoff"))) {
                                  return true;
                              } else {
                                  return false;
                              }
                          }
                      case '4':
                          if (actToken.length() != 4) {
                              return false;
                          } else {
                              actToken = st.nextToken();
                              if (actToken.equals("quit")) {
                                  return true;
                              } else {
                                  return false;
                              }
                          }
                      default:
                          return false;
                  }
              }
          }
      }
```

Diese Methode prüft den ihr übergebenen String auf Korrektheit bezüglich des definierten Formates. Zum Zerteilen des Strings in einzelne Worte wird ein `StringTokenizer` eingesetzt, der bei den üblichen Whitespace-Zeichen wie Leerzeichen, Zeilenumbruch etc. trennt. Für eine korrekte Anfrage muss der String zunächst einmal aus drei Worten bestehen. Das erste davon muss `request` lauten, das zweite muss mit einer `1`, `2`, `3` oder `4` beginnen und genau vier Zeichen lang sein (und damit eine korrekte ID für Weiche, Signal oder Sensor sein), das dritte muss eines der Schlüsselworte für den angeforderten Zustand sein (dieser Zustand muss zu dem Grundtyp, d.h. Weiche, Signal oder Sensor passen). Es wird nicht geprüft, ob ein Element mit der genannten ID überhaupt im simulierten Gleisnetz existiert, sondern nur, ob die Anfrage in der Form korrekt ist.
Von der gleichen Bauart ist die Funktion `initRequest`, welche daher hier nicht aufgeführt wird.

```
private synchronized void addRequest(String request) {
    requestList.add(request);
```

```
}

public synchronized boolean checkRequestPending() {
    if (requestList.size() == 0) return false;
    else return true;
}

public synchronized String getRequest() {
    String retvalue = new String();
    retvalue = (String)requestList.removeFirst();
    return retvalue;
}
```

Diese drei Methoden realisieren den Zugriff auf die Liste der Anfragen. Diese Zugriffe müssen getrennt voneinander erfolgen, um typische Nebenläufigkeitsprobleme zu umgehen, daher sind die Methoden `synchronized`.
`addRequest`, das Hinzufügen einer neuen Anfrage, muss nur von diesem Server selbst aufgerufen werden, daher kann sie `private` deklariert werden. `checkRequestPending` prüft, ob aktuell Anfragen in der Liste vorhanden sind, `getRequest` liefert die erste (und damit älteste) Anfrage zurück. Die beiden zuletzt genannten Methoden werden vom Simulator aufgerufen und sind daher als `public` deklariert.

```
private synchronized void addDriver(String request,
  SocketChannel driver) {
    ArrayList temp = new ArrayList();
    StringTokenizer st = new StringTokenizer(request, " " + '\n');
    String actToken = st.nextToken();
    actToken = st.nextToken();
    temp.add(new Integer(actToken));
    temp.add(driver);
    driverList.add(temp);
}

public synchronized boolean checkDriverPending() {
    if (driverList.size() == 0) return false;
    else return true;
}

public synchronized ArrayList getDriver() {
    ArrayList retval = new ArrayList();
    retval = (ArrayList)driverList.removeFirst();
```

```
        return retval;
    }
```

Diese Methoden realisieren den Zugriff auf die Liste der Treiberinformationen, die jedes Mal gefüllt wird, wenn ein Treiber sich erstmals anmeldet. Es werden dann zwei Werte, als da wären die ID des von diesem Treiber gesteuerten HW-Elements und der SocketChannel, auf dem der Treiber angebunden ist, in dieser Liste abgelegt. Der Simulator prüft, bevor die eigentliche Simulation angelaufen ist, ständig, ob sich neue Treiber angemeldet haben und teilt dies dem `StatusSender` mit, da dieser diese Daten benötigt.

7.8.5.5 Richtung Simulator - Treiber

Die Gegenrichtung wird relativ ähnlich realisiert. Wir haben dieses Mal einen Thread, der ständig den Status einer Liste überwacht, und sobald ein String in dieser Liste vorhanden ist, diesen an den Hardware-Treiber sendet. Auch hier gelten die oben gemachten Anmerkungen zum Quellcode (keine Kommentare, nicht vollständig aufgeführt).

```
public class StatusSender extends Thread {
    private ArrayList statusList;
    private HashMap drivers;
    private Logger logger;
```

Als Variable verfügt diese Klasse wiederum über eine Liste, in der in diesem Fall die Statusmeldungen für den SI gesammelt werden. In einer HashMap werden die Daten über die Treiberinstanzen festgehalten, als Index dienen die IDs der jeweiligen Gleisnetzelemente. Zudem steht auch der Logger zur Verfügung.

```
public void run() {
    String report;
    while (true) {
        if (checkPending()) {
            report = getReport();
            StringTokenizer st = new StringTokenizer(report, " ");
            try {
                String hw = st.nextToken();
                hw = st.nextToken();
                SocketChannel s = (SocketChannel)
                  drivers.get(new Integer(hw));
                if (!s.isConnected()) {
                    drivers.remove(new Integer(hw));
                    s = null;
                }
```

```
                    if (s != null) {
                        try {
                            report = report + "\n";
                            CharBuffer tmpbuf =
                              CharBuffer.allocate(report.length()).put
                              (report);
                            tmpbuf.rewind();
                            Charset enc = Charset.forName("US-ASCII");
                            ByteBuffer bb = enc.encode(tmpbuf);
                            int len = s.write(bb);
                            if (len <= 0) {
                                System.out.println("write returned "
                                  + len);
                                addAgain(report);
                            } else {
                                logger.info("Writing " + report);
                            }
                        } catch (Exception e) {
                            logger.info("Failed writing "+ report);
                            addAgain(report);
                        }
                    } else {
                        logger.info("Cannot send report '" + report +
                          "' - Driver unknown");
                    }
                } catch (Exception e) {
                    logger.info("Failed writing (do not try again) "
                      + report);
                    e.printStackTrace();
                }
            } else {
                try {
                    Thread.sleep(100);
                } catch (Exception e) {
                }
            }
        }
    }
```

Der Ablauf erfolgt auch hier in einer Endlosschleife:
Es wird geprüft, ob eine Meldung in der Liste vorliegt – ist das der Fall, wird ver-

sucht, den zuständigen Treiber zu ermitteln und an diesen die Meldung abzuschicken. Wird kein Treiber für das angesprochene HW-Element gefunden, wird die Nachricht verworfen, scheitert etwas beim Sendevorgang selbst, wird die Meldung erneut der Liste hinzugefügt, damit sie nicht verloren geht und erneut ein Sendeversuch unternommen werden kann.
Anschließend kehrt die Endlosschleife an den Beginn zurück.

```
public synchronized boolean checkPending() {
    if (statusList.size() == 0) return false;
    else return true;
}

private synchronized void addAgain(String report) {
    statusList.addFirst(report);
}

private synchronized String getReport() {
    String retvalue = new String();
    retvalue = (String)statusList.removeFirst();
    return retvalue;
}


public synchronized void addReport(String report) {
    statusList.add(report);
}
```

Dieses sind die Zugriffsmethoden auf die Warteliste – in diesem Fall darf nur der Sendethread selbst Strings entnehmen und im Falle des gescheiterten Sendens erneut einfügen, daher haben `getReport` und `addAgain` nur `private` Zugriff. Das Hinzufügen (`addReport`) und Prüfen auf nicht-leere Liste (`checkPending`) ist `public` deklariert.
Zu beachten ist noch, dass keine Prüfung auf Korrektheit des Strings erfolgt, der Sender geht also davon aus, dass das Format bereits anderweitig, also im Simulator, der ja die Strings generiert, sichergestellt wurde.

```
public synchronized void addDriver(ArrayList drv) {
    Integer hw = (Integer) drv.get(0);
    SocketChannel sock = (SocketChannel) drv.get(1);
    drivers.put(hw, sock);
    System.out.println("Adding driver for " + hw);
}
```

```
public synchronized void removeDriver(Object hw) {
    if (hw instanceof Hardware) {
        drivers.remove(new Integer(((Hardware)(hw)).
          getSIid()));
    } else if (hw instanceof Route) {
        drivers.remove(new Integer(((Route)(hw)).getSIid()));
    } else {
        System.out.println("Unknown remove command");
    }
}

protected void clearDrivers() {
    this.drivers.clear();
}
```

Mittels dieser Methoden greift der Simulator auf die Treiber-HashMap zu, um entweder neue Treiber einzutragen, einzelne Treiberdaten zu löschen (wenn der Treiber ein Exit-Request gesendet hat) oder alle Treiberdaten zu löschen (wenn eine neue Simulation begonnen wird).

7.8.5.6 Treiber

Es wurden mehrere Treiber implementiert, um den Simulator an den SI anbinden zu können (für jede Hardwareklasse - Sensor, Signal, Weiche, Kreuzungsweiche - einer. Das Funktionsprinzip dieser Treiber ist quasi identisch, weswegen hier stellvertretend auch für die anderen Treiber der Weichentreiber gezeigt werden soll.
Die Treiber wurden in C implementiert.
Es müssen fortlaufend zwei Schnittstellen beobachtet werden: Zum einen ist das das Shared Memory, zum anderen ein Socket. Tritt eine Änderung im SHM auf, muss eine Textmeldung an den Simulator generiert und versendet werden, die dann von der oben (7.8.5.4 auf Seite 393) beschriebenen Schnittstelle aufgenommen und im Simulator umgesetzt wird. Umgekehrt kann eine Statusmeldung beim Treiber eingehen, nach der er dann den Zustand des SHM verändern muss.
Zur Bedeutung der einzelnen SHM-Belegungen und zu den vom Treiber bei Vorfinden bestimmter Belegungen geforderten Reaktionen sei an dieser Stelle auf den entsprechenden Teil der SI-Spezifikation verwiesen. Es folgen nun die relevanten Auszüge aus dem Code des Weichentreibers.

```
static int driver_point_state_change(struct driver_info *drv,
  u_int32_t in, u_int32_t *out) {
```

```
        char send_buffer[64];
        char idbuf[6];

        sprintf(idbuf, " %d ", drv->id);

        if ((in == (1<<POINT_SET_STRAIGHT)) &&
          (drv->type != TYPE_POINT_LR)) {
            *out = (drv->old_out & OUTPUT_MASK) | in;
            strcpy(send_buffer, STR_REQUEST);
            strcat(send_buffer, idbuf);
            strcat(send_buffer, STR_STRAIGHT);
            strcat(send_buffer, "\n");
            send_msg(drv->private_data, send_buffer);
        } else if ((in == (1<<POINT_SET_LEFT)) &&
          (drv->type != TYPE_POINT_SR)) {
            *out = (drv->old_out & OUTPUT_MASK) | in;
            strcpy(send_buffer, STR_REQUEST);
            strcat(send_buffer, idbuf);
            strcat(send_buffer, STR_LEFT);
            strcat(send_buffer, "\n");
            send_msg(drv->private_data, send_buffer);
        } else if ((in == (1<<POINT_SET_RIGHT)) &&
          (drv->type != TYPE_POINT_SL)) {
            *out = (drv->old_out & OUTPUT_MASK) | in;
            strcpy(send_buffer, STR_REQUEST);
            strcat(send_buffer, idbuf);
            strcat(send_buffer, STR_RIGHT);
            strcat(send_buffer, "\n");
            send_msg(drv->private_data, send_buffer);
        } else if (in == (1<<POINT_SET_QUIT)) {
            *out = in | (1<<POINT_GET_OFF);
            strcpy(send_buffer, STR_REQUEST);
            strcat(send_buffer, idbuf);
            strcat(send_buffer, STR_QUIT);
            strcat(send_buffer, "\n");
            send_msg(drv->private_data, send_buffer);
        } else if (in == 0) {
            *out = (drv->old_out & OUTPUT_MASK) | in;
        } else {
    printf("invalid request\n");
            *out = in | (1<<POINT_GET_INVALID);
```

```
            strcpy(send_buffer, STR_REQUEST);
            strcat(send_buffer, idbuf);
            strcat(send_buffer, STR_QUIT);
            strcat(send_buffer, "\n");
            send_msg(drv->private_data, send_buffer);
        }
        return 0;
    }
```

Diese Funktion wird dann vom SI-Treiberframework aufgerufen, wenn eine Anweisung an das Element übermittelt werden soll. In der Variablen `in` ist der Eingabebereich des Shared Memory enthalten, der also die Anweisung umfasst. Es wird dann geprüft, ob es sich um eine korrekte Anforderung gemäß SI-Treiberspezifikation handelt. Ist dem so, wird eine Meldung zusammengesetzt und mittels der Hilfsfunktion `send_msg()` abgeschickt. In `out` wird anschließend die SHM-Belegung zurückgeliefert (die Schaltanweisung ist an dieser Stelle natürlich noch nicht ausgeführt). Bei einer inkorrekten Anforderung setzt der Treiber ein „invalid“-Bit und meldet sich beim Simulator ab.

```
static int driver_point_callback(struct driver_info *drv,
  u_int32_t in, u_int32_t *out) {

    int bytes_read;
    int bytes_worked;

    int new_shm;

    struct internal_data *data = drv->private_data;
    char *read_buffer = data->cmd;
    if (!recv_msg(data)) {
      *out = (drv->old_out & OUTPUT_MASK) | in;
    } else {
      bytes_read = strlen(read_buffer);
      read_buffer[bytes_read] = '\0';
            bytes_worked = 0;
            if (strncmp(read_buffer, STR_STATUS, 7) == 0) {
                bytes_worked += 7;
                if (atoi(read_buffer+bytes_worked) == drv->id) {
                    bytes_worked += 4;
                    if (strncmp(read_buffer+bytes_worked,
                      STR_STRAIGHT, 9) == 0) {
                        new_shm = in & INPUT_MASK;
```

```
                        new_shm += (1<<POINT_GET_STRAIGHT);
                        *out = new_shm;
                        drv->old_out = new_shm;
                    } else if (strncmp(read_buffer+bytes_worked,
                      STR_LEFT, 5) == 0) {
                        new_shm = in & INPUT_MASK;
                        new_shm += (1<<POINT_GET_LEFT);
                        *out = new_shm;
                        drv->old_out = new_shm;
                    } else if (strncmp(read_buffer+bytes_worked,
                      STR_RIGHT, 6) == 0) {
                        new_shm = in & INPUT_MASK;
                        new_shm += (1<<POINT_GET_RIGHT);
                        *out = new_shm;
                        drv->old_out = new_shm;
                    } else if (strncmp(read_buffer+bytes_worked,
                      STR_UNDEFINED, 10) == 0) {
                        new_shm = in & INPUT_MASK;
                        *out = new_shm;
                        drv->old_out = new_shm;
                    } else if (strncmp(read_buffer+bytes_worked,
                      STR_FAILURE, 8) == 0) {
                        new_shm = in & INPUT_MASK;
                        new_shm += (1<<POINT_GET_FAILURE);
                        *out = new_shm;
                        drv->old_out = new_shm;
                    } else {
                        *out = (drv->old_out & OUTPUT_MASK) | in;
                    }
                } else {
                    *out = (drv->old_out & OUTPUT_MASK) | in;
                }
            } else {
                *out = (drv->old_out & OUTPUT_MASK) | in;
            }
        }

        return 0;
    }
```

Diese Funktion nimmt Nachrichten des Simulators an, wertet sie aus, und falls sie korrekt

gemäß dem definierten Format zusammengesetzt sind, wird der SHM-Zustand (bzw. dessen Kopie in `out`) entsprechend gesetzt. Sind die Nachrichten nicht korrekt, werden sie verworfen und nicht weiter beachtet. Diese Funktion wird vom Treiberframework des SI periodisch aufgerufen.

```
static int driver_point_init(struct driver_info *drv) {

    struct internal_data *data = (struct internal_data *)
      malloc(sizeof(struct internal_data));

    char port[6];
    char send_buffer[64];

    struct sockaddr_in *sender_addr = (struct sockaddr_in *)
      malloc(sizeof(struct sockaddr_in));

    sender_addr->sin_family = PF_INET;
    sender_addr->sin_addr.s_addr = inet_addr(SIM_HOST);
    sender_addr->sin_port = htons(SIM_PORT);

    data->sender_addr = sender_addr;
    init_con(data);
    strcpy(send_buffer, STR_INIT);
    sprintf(port, " %d", drv->id);
    strcat(send_buffer, port);

    strcat(send_buffer, "\n");

    drv->private_data = data;

    send_msg(drv->private_data, send_buffer);

    return 0;
}
```

In der Init-Funktion wird die Verbindung zum Simulator hergestellt und die Init-Meldung an diesen geschickt. Daher muss der Simulator zum Zeitpunkt des Treiber-Starts bereits laufen – der Simulator als Prozess, nicht die eigentliche Simulation.

Die Treiber nutzen insgesamt drei Hilfsfunktionen, die mit `send_msg`, `recv_msg` und `init_con` bezeichnet werden. Diese tun im Wesentlichen das, was ihr Name andeutet:

`send_msg` sendet einen String über den offenen Socket ab, `recv_msg` überträgt auf dem Socket eingegangene Daten in einen Lesepuffer und extrahiert daraus komplette Strings, die dann vom Treiber ausgewertet werden können, `init_con` baut die Verbindung auf. Die weiteren Treiber unterscheiden sich nicht wesentlich von diesem, die Differenzen liegen lediglich darin, dass andere Elementtypen angesteuert werden, daher die SHM-Belegungen andere Bedeutungen haben und folglich andere Textmeldungen ausgetauscht werden.
Die Treiber wurden ursprünglich ohne einen vorliegenden SI entwickelt. Dies hatte zur Folge, dass erst bei Existenz eines lauffähigen SI – kurz vor dem Projekttag – wirkliche Tests betrieben werden konnten. Dabei zeigte sich, dass es besser war, für jeden Treiber eine Verbindung zum Simulator aufzubauen und diese dann für den Betrieb aufrechtzuerhalten, als bei jeder Meldung eine neue Verbindung zu initiieren. Die hierfür nötigen Umstellungen wurden von der SI-Gruppe vorgenommen.

Silvia Graumann, Niklas Polke, Arne Stahlbock

7.8.6 Tests

In diesem Kapitel soll beschrieben werden, welche Tests an Teilkomponenten und der gesamten Simulatorsoftware durchgeführt wurden.

7.8.6.1 Tests von Methoden

Tests von Methoden sind die unterste Ebene der am Simulator durchgeführten Tests, sie wurden meist derart durchgeführt, dass in die Methoden zwischenzeitlich `println`-Anweisungen eingebaut wurden, die zur Beobachtung von Variablen oder des Kontrollflusses dienten. Im Wesentliche wurde dies an den mit der Fortbewegung der Bahnen zusammenhängenden Methoden betrieben (Positionsberechnung, Signalvorausschau).
Eine andere Vorgehensweise konnte einfach bei den Zeichenmethoden der GUI beschritten werden, hier wurden einfach die Zeichenmethoden alle, bis auf eine für eine bestimmte HW-Klasse, auskommentiert, so dass nur Elemente einer Klasse gezeichnet wurden und so die jeweilige Zeichenmethode geprüft werden konnte.

7.8.6.2 Modultests

Diese Ebene von Tests ist innerhalb der Testreihen als die größte zu bezeichnen. Hier wurden verschiedene Module (bestehend aus mehreren Funktionen im Zusammenspiel) auf ihre gewünschte Funktionalität getestet.
Dies sind:

- GUI: Visualisierung der Elemente
  Werden alle HW-Elemente richtig in Bezug auf Ort und Zustand dargestellt?

- GUI: Visualisierung der Bahn
  Als komplexester Unterpunkt der Visualisierung wurde die Bahn gesondert untersucht.

- GUI: Menüfunktionen
  Bewirken die Menübefehle das, was sie sollen?

- Parser: Einlesen
  Mit verschiedenen TNDs wurde getestet, ob der Einlesevorgang inkl. des Aufbauens der Objektstruktur richtig funktioniert. Die Prüfung dafür wurde mit Hilfe der GUI vorgenommen (richtige Darstellung lässt auf richtiges Einlesen schließen).

- Parser: Inkorrekte TND
  TND-Dateien, die nicht der Sprachdefinition entsprechen, müssen abgelehnt werden.

- Bahnen: Fahrt
  Fährt die Bahn auf normaler Strecke (keine Weichen, Kreuzungen, Signale), wie sie soll?

- Bahnen: Fahrt über Weichen
  Fährt die Bahn richtig über Weichen? D.h. fährt sie am richtigen Ende von der Weiche herunter? Schaltet sie passive Weichen beim Auffahren? Entgleist sie beim spitzen Befahren, wenn die Weiche im undefinierten Zustand ist? Wird Auffahren auf nicht auffahrbare Weichen festgestellt?

- Bahnen: Fahrt über Kreuzungen
  Fährt die Bahn richtig über eine Kreuzung?

- Bahnen: Fahrt über Sensoren
  Bahnen fahren über Sensoren und es wird anhand des Strings, der an die Treiber gesendet werden soll, für jede Sensorart geprüft, ob der richtige Output erfolgt.

- Bahnen: Beachten von Signalen
  Halten Bahnen vor Stop-Signalen? Halten sie auch, falls eine andere als die für ihre Route benötigte Richtung freigegeben ist? Wird die Sichtweite eingehalten (d.h. wenn ein Signal erst ganz spät auf Stop schaltet, fährt die Bahn dann noch durch?)? Funktioniert die Vorausschau über mehrere Elemente?

- Bahnen: Nicht auffahren
  Wenn eine Bahn hinter einer anderen herfährt, darf sie nicht auffahren.

- Bahnen: Kollisionen
  Beide Kollisionstypen werden inszeniert, der Simulator muss das Auftreten feststellen. Insbesondere sind hier auch die Kreuzungsweichen zu beachten und der Fall, dass eine Bahn im einem Zyklus schon ein Element verlassen hat, eine andere dann auf dieses Element gelangt, was noch als Kollision gewertet werden muss.

- Rückfallweichen
  Schalten sie nach Überfahren in die Standardlage zurück?

- Weichen und Signale: Schalten
  Werden Schaltanforderungen umgesetzt? Werden unmögliche Anweisungen bemerkt? Wird die Switchtime beachtet (nicht quantitativ getestet, sondern durch Abschätzen von Größenordnungen)?

- Treiber: Reaktion auf Eingaben
  Den Treibern wurden Eingaben vom Simulator geschickt, es wurde eine Veränderung des SHM-Zustandes erwartet. Umgekehrt wurden SHM-Zustände geändert, die das Aussenden einer Meldung an den Simulator provozieren sollten. Ebenso musste die Initialisierung und das Verhalten auf inkorrekte Eingaben getestet werden. Hinweis: diese Tests sind mit gewisser Vorsicht zu genießen, da sie zwar prinzipiell an den in den Treibern verwendeten Algorithmen durchgeführt wurden, die Treiber aber, um ohne SI lauffähig zu sein, in gewissem Rahmen verändert wurden. In Realität sollen die Treiber nicht direkt auf dem SHM arbeiten, sondern lediglich auf Kopien, die vom Treiberframework des SI geliefert werden, im Test wurde auf ein SHM direkt zugegriffen.

- Kommunikation mit den Treibern
  `DriverRequestServer` und `StatusSender` im Test.

7.8.6.3 Gesamttests

Gesamttests konnten lange Zeit nur in der Form durchgeführt werden, dass ein Gleisnetz mit mehreren Bahnen betrieben wurde und die Anforderungen entweder per telnet-Konsole an den Simulator geschickt wurden (sprich die Strings, die sonst von den treibern kommen, direkt eingeben) oder man ein HW-Element mit Treiber ansteuerte, wozu man dann ein auf die Schnelle zusammengesetztes Kommandozeilen-Tool zur Änderung von SHM-Werten einsetzte. Erst mit Vorliegen eines lauffähigen SI kurz vor dem Projekttag konnte der Simulator als Komplettsystem (bzw. das Zusammenspiel Simulator - SI) getestet werden. Hier wurden vor allem Dauertests durchgeführt, d.h. das kombinierte System für mehrere Stunden ohne menschlichen Eingriff betrieben.

7.8.6.4 Beispieltest

An dieser Stelle soll beispielhaft für andere ein Test aus der Reihe der Modultests detaillierter beschrieben werden. Es handelt sich um einen Test zur Prüfung des Beachtens von Signalen.
Es wurde folgende TND eingesetzt:

```
definitions {
    rr-sensors: g1;
    tg-sensors: g2, g3;
    s-signals: s1;
    routes: r1;
}

coordinates {
    g1:(100,100,0);
    g2:(20000,100,0);
    g3:(50000,100,0);
    s1:(20000,1000,0);
}

signal-properties {
    s1: rr-straight switchtime 100;
}

relations {
    g1 A: entrance;
    g1 B: g2 A;
    g2 B: g3 A;
    g3 B: exit;
}

signal-positions {
    s1: g2 A;
}

routedefinitions {
    r1: g2, g3 - request-at: g1;
}

conditions { }
```

```
clearances {
    r1: s1 straight;
}

hardwaremap { }
```

Hier wird eine einfache, in nur eine Richtung befahrbare Strecke über drei Sensoren definiert. Am zweiten davon befindet sich ein Signal.
Der Simulator wird nun mit dieser TND beschickt und eine vorkonfigurierte Simulation gewählt, bei der das Signal im undefinierten Zustand belassen wird. Die Zahl der Bahnen wird auf eine beschränkt, diese soll die einzige mögliche Route befahren. Erwartet wird, dass die Bahn am Signal anhält.

Das Logfile enthält folgenden Output (nur Auszug der relevanten Stellen):

```
INFO: Tram Tram: Wrong signal percepted in1. iteration
[...]
INFO: Trying to move 138.80043 steps
INFO: Tram Tram0 is now located:
Head on TE 1 between HW g1 and HW g2 at distance 19570
Tail on TE 1 between HW g1 and HW g2 at distance 16371
INFO: Time: 1111182650317
INFO: Trying to move 139.60043 steps
INFO: Tram Tram0 is now located:
Head on TE 1 between HW g1 and HW g2 at distance 19709
Tail on TE 1 between HW g1 and HW g2 at distance 16510
INFO: Time: 1111182650427
INFO: Trying to move 139.40044 steps
INFO: Tram Tram0 is now located:
Head on TE 1 between HW g1 and HW g2 at distance 19848
Tail on TE 1 between HW g1 and HW g2 at distance 16649
INFO: Time: 1111182650537
INFO: Trying to move 139.20044 steps
INFO: Tram Tram0 must stop in front of signal while 89.20044 steps
  are pending
INFO: Tram Tram0 must stop in front of signal while 0.0 steps
  are pending
INFO: Tram Tram0 is now located:
Head on TE 1 between HW g1 and HW g2 at distance 19898
Tail on TE 1 between HW g1 and HW g2 at distance 16699
INFO: Time: 1111182650647
INFO: Trying to move 138.8 steps
```

```
INFO: Tram Tram0 must stop in front of signal while 138.8 steps
  are pending
INFO: Tram Tram0 must stop in front of signal while 0.0 steps
  are pending
INFO: Tram Tram0 is now located:
Head on TE 1 between HW g1 and HW g2 at distance 19898
Tail on TE 1 between HW g1 and HW g2 at distance 16699
INFO: Time: 1111182650757
INFO: Trying to move 138.8 steps
INFO: Tram Tram0 must stop in front of signal while 138.8 steps
  are pending
INFO: Tram Tram0 must stop in front of signal while 0.0 steps
  re pending
INFO: Tram Tram0 is now located:
Head on TE 1 between HW g1 and HW g2 at distance 19898
Tail on TE 1 between HW g1 and HW g2 at distance 16699
```

Wie man sehen kann, fährt die Bahn zunächst noch weiter, bis das Signal erreicht ist und hält dort an. Sie versucht dann, jeden Zyklus wieder weiterzufahren, was aber, da das Signal nicht umschaltet, nicht gelingt.
Wird die Situation neu gestartet, diesmal mit dem Signal in der gewünschten Stellung, so ist der Output:

```
INFO: Tram Tram0 is now located:
Head on TE 1 between HW g1 and HW g2 at distance 19848
Tail on TE 1 between HW g1 and HW g2 at distance 16649
INFO: Time: 1111183956307
INFO: Trying to move 139.20044 steps
INFO: Tram Tram0: Correct signal percepted in1. iteration
INFO: Tram Tram0 is now located:
Head on TE 2 between HW g2 and HW g3 at distance 87
Tail on TE 1 between HW g1 and HW g2 at distance 16788
INFO: Report to SI: status g2 toggleon
INFO: Cannot send report 'status g2 toggleon' - Driver unknown
INFO: Time: 1111183956417
INFO: Trying to move 139.00044 steps
INFO: Tram Tram0 is now located:
Head on TE 2 between HW g2 and HW g3 at distance 226
Tail on TE 1 between HW g1 and HW g2 at distance 16927
```

Der Kopf der Bahn überfährt also das Element, löst den Sensor aus (da kein im Test kein Treiber angeschlossen war, kann auch keine Meldung erfolgen) und bewegt sich weiter.

7.8.6.5 Testergebnisse

Bei den verschiedenen Tests wurde eine Reihe von Fehlern im Programm gefunden, die dann auch behoben werden konnten. Momentan sind keine weiteren Fehler bekannt, was natürlich nicht automatisch heißt, dass es keine mehr gibt.
Vor allem nach Anschluss des SI traten noch einmal einige Fehler zu Tage, die erst im längeren Betrieb erkennbar waren (und längerer Betrieb war ohne SI quasi nicht möglich, es hätte dann ein Mensch ständig als Ersatz-SI agieren und via telnet Kommandos an den Simulator schicken müssen). Auch diese Fehler konnten zeitnah behoben werden.
Da das System im Vorfeld des Projekttages und auch an diesem selbst jeweils über eine Dauer von mehreren Stunden problemlos lief und erst durch Befehl des Bedieners gestoppt wurde, darf mit relativ hoher Sicherheit angenommen werden, dass keine gravierenden Fehler mehr vorhanden sind.

Arne Stahlbock

7.8.7 Anleitung

Dieses Kapitel enthält die Bedienungsanleitung für den Simulator.

7.8.7.1 Voraussetzungen und Installation

7.8.7.1.1 Systemvoraussetzungen Die Software ist für den Einsatz auf einem handelsüblichen PC konzipiert. Besondere Mindestanforderungen an die CPU-Geschwindigkeit oder die Größe des Hauptspeichers haben sich bisher nicht bestimmen lassen, da die Testeinsätze auch auf den langsamsten geprüften Rechnern (Dual Pentium II 350 MHz, 128 MB RAM bzw. Pentium III 866 MHz, 128 MB RAM) noch zufriedenstellend verliefen. Verpflichtend ist jedoch eine Netzwerkanbindung.
Als Betriebssystem wird ein gängiges Linux empfohlen (getestet Debian 3.0 und 3.1, SuSE 9.0), wenngleich sicher auch andere möglich sind. Die oben verlangte Netzwerkanbindung muss über das TCP/IP-Protokoll verfügbar sein. Ebenso ist, da der Simulator eine Grafikanwendung ist, eine grafische Benutzeroberfläche wie bspw. KDE oder Gnome Voraussetzung.
Auf dem System muss folgende Software verfügbar sein:

- Java™-Compiler
- Java™-Interpreter
- JFlex
- CUP Parser Generator for Java™
- C-Compiler (für die Treiber)

Java™-Compiler und -Interpreter gibt es zahlreich, während der Entwicklung des Simulators wurde Java™2 Platform, Standard Edition, v 1.4.2 von Sun™, eingesetzt, weswegen wir an dieser Stelle auch genau dieses empfehlen, erhältlich unter
http://java.sun.com/j2se/1.4.2/download.html
Andere mögen auch funktionieren, das kann an dieser Stelle aber nicht garantiert werden.

JFlex (eingesetzte Versionen: 1.3.5 und 1.4.1) kann unter
www.jflex.de
bezogen werden.

CUP (eingesetzte Version 0.10k) kann unter
http://www.cs.princeton.edu/ appel/modern/java/CUP/
bezogen werden.

Als C-Compiler wurde gcc in den Versionen 2.95.4 und 3.3.5 eingesetzt
(http://gcc.gnu.org).

7.8.7.1.2 Installation Das Sourceverzeichnis des Simulators (von nun an als `src` bezeichnet) ist in ein beliebiges Verzeichnis auf dem Zielrechner zu platzieren. Dabei darf die Verzeichnis- und Dateistruktur innerhalb von `src` nicht geändert werden.
Das bei der CUP-Installation enthaltene Archiv cup.jar ist ebenso wie das Verzeichnis `src` in die Umgebungsvariable CLASSPATH aufzunehmen.
Java™-Compiler und -Interpreter sowie JFlex nach den zugehörigen Anleitungen installieren, falls nicht ohnehin eine Installation über ein in der verwendeten Distribution enthaltenes Paket gewählt wird.

Der Simulator kann nun kompiliert werden, indem nach `src` gewechselt wird und `make` bzw. `make sim` ausgeführt wird.
Sollen die Treiber kompiliert werden, ist `make drivers` aufzurufen, alles zusammen ist mit `make all` zu erreichen.
Nach erfolgreichem Kompilieren wird der Simulator mit dem Shellscript `SimTracs.sh` gestartet.

7.8.7.2 Ablauf einer Simulation

Nach erfolgtem Start findet der Benutzer ein Fenster mit zunächst leerer (d.h. grauer) Darstellungsfläche vor.
Über `Main Menu` – `Load TND` gelangt man zu einem Dateiauswahldialog, in dem man eine TND-Datei auswählt, die geladen werden soll. Nach der Auswahl baut sich umgehend das Gleisnetz auf der Darstellungsfläche auf.

War die geladene Datei nicht im korrekten Format, erscheint eine Fehlermeldung. Der Bediener muss dann eine neue Datei laden.
Wir gehen an dieser Stelle aber vom erfolgreichen Laden aus, ein Gleisnetz ist auf der Darstellungsfläche zu sehen. Nun hat der Bediener die Möglichkeit, zwischen zwei Optionen zu wählen:

- nur Anzahl der Bahnen vorgeben
- detaillierte Konfiguration aller Gleisnetzelemente und Bahnen

Möglichkeit 1 ist erreichbar über `Options` – `Configure Random Simulation`, während sich die Detailkonfiguration unter `Options` – `Configure Preset Simulation` verbirgt.

Wir nehmen an dieser Stelle die detaillierte Konfiguration. Es erscheint ein Fenster zur Bestimmung der Bahnanzahl. Gefolgt wird es von einem Dialog, in dem Eigenschaften der Bahnen eingegeben werden können, danach erscheint der Dialog für die Weicheneinstellungen, annschließend die Sensoreinstellungen, zum Schluss die Signaleinstellungen. Nähere Beschreibungen der verfügbaren Einstellungen finden sich im folgenden Abschnitt.

Entscheidet sich der Nutzer dafür, nur die Anzahl der Bahnen vorzugeben, muss er nur den ersten der fünf Dialoge absolvieren.

Ist die Konfiguration abgeschlossen, ist der Zeitpunkt gekommen, den ggf. angeschlossenen Steuerinterpreter und die Treiber zu starten. Ein Anbinden des SI während einer laufenden Simulation erscheint nicht ratsam, da der SI dann potentiell bereits eine kritische und nicht mehr behebbare Situation vorfinden könnte.

Anschließend erfolgt der Start der Simulation über `Options` – `Start Simulation`.

Nun beginnen die ersten Bahnen in das Gleisnetz einzutreten und ihre Routen zu absolvieren. Sensoren geben Rückmeldungen, Signale und Weichen nehmen Stellanweisungen an. Ist für ein Hardware-Element kein Treiber angemeldet, so gibt dieses keine Rückmeldungen ab, es wird dann nur im Simulationslog festgehalten, dass eine Rückmeldung mit einem bestimmten Text erfolgt wäre, aber nicht gesendet werden konnte, da kein Treiber bekannt war, an den man hätte senden sollen. Anweisungen an die Elemente kann man hingegen auch geben, wenn für das Element kein Treiber aktiv ist. Man benutze hierfür eine Telnet-Konsole, die Verbindungsanfrage ist an Port 12345 auf dem Simulatorrechner zu richten. Nach Verbindungsaufbau kann dann ein Request-String eingegeben werden, nach Absenden des Befehls beendet sich die Verbindung wieder. Zu beachten ist, dass diese Hintertür zur Befehlseingabe auch für die Elemente funktioniert, für die ein Treiber angemeldet ist. Der Treiber bzw. der SI bekäme in dem Fall

nicht direkt mit, dass eine Anweisung gegeben wird, sondern würde erst eine für ihn unerwartete Rückmeldung, dass eine Schaltung erfolgt sei, erhalten. Es wird daher nicht empfohlen, diese Art der Eingabe für solche Elemente zu nutzen, für die der SI gerade zuständig ist.

Mögliche Eingaben sind:

- für Weichen: `request <ID> <state>`, wobei `<ID>` der vom Compiler vergebene Bezeichner (nicht der aus der TND!) für die Weiche ist, `<state>` eine von drei möglichen Stellungen: `straight`, `right`, `left`
- für Signale: `request <ID> <state>`, wobei `<ID>` der vom Compiler vergebene Bezeichner für das Signal ist, `<state>` eine der möglichen Stellungen: `straight`, `right`, `left`, `stop`, `waitstraight`, `waitright`, `waitleft`, `rrstraighton`, `rrstraightoff`, `rrrighton`, `rrrightoff`, `rrlefton`, `rrleftoff`

Die RR-Anweisungen beziehen sich auf das An- bzw. Ausschalten der RR-Anzeigen für die jeweiligen Richtungen, alle anderen Befehle dürften selbsterklärend sein. Wird eine Anweisung an eine Weiche oder ein Signal gegeben, die dieses Element nicht erfüllen kann (z.B. weil es die entsprechende Anzeige nicht hat oder nicht für die verlangte Richtung zuständig ist), so führt das zu einem Fehler, den der Simulator erkennt und genauso behandelt, wie er unmögliche Schaltbefehle seitens des SI behandelt.

Im Normalfall läuft die Simulation nun so lange, bis der Benutzer sie beendet. Sie kann unterbrochen und unmittelbar an gleicher Stelle fortgesetzt werden. Vorsicht bei Stopps mit angeschlossenem SI: Es werden während der Pause natürlich keine Stellanweisungen umgesetzt, was Auswirkungen auf den Ablauf im SI haben kann. Jedoch läuft die Zeit auch in Pausen weiter, so dass nach der Pause umgehend alle Stellanweisungen umgesetzt werden, die in der Zeit der Pause hätten umgesetzt werden müssen. Stellbefehle, die in der Pause eingehen, werden als direkt nach der Pause eingegangen betrachtet, was entsprechende Auswirkungen auf den Zeitpunkt ihrer Umsetzung hat.

Verlässt eine Bahn das Netz und wird dadurch die konfigurierte Zahl an Bahnen unterschritten, wird eine neue Bahn in das Netz einfahren. Diese wird bei einer detailliert konfigurierten Simulation die gleiche Route befahren wie die verschwundene Bahn.

Neben dem Normalfall gibt es natürlich auch noch den Fehlerfall, der zum Abbruch der Simulation führt. Eine so gestoppte Simulation kann auch nicht fortgesetzt werden, es muss in dem Fall eine neue Simulation begonnen werden.
Erkannte Fehlerfälle sind:

- Es tritt eine Kollision auf

- Es wird eine Stellanweisung gegeben, die vom angesprochenen Element „hardwareseitig“ nicht umsetzbar ist.
- Eine Bahn wird von der spitzen Seite auf eine Weiche geleitet, die sich im Schaltzustand `undefined` befindet. (Dieser Zustand kann allenfalls am Anfang auftreten, wenn für das Element noch keine Schaltanweisung gegeben worden ist.)
- Eine Weiche wird umgestellt, während eine Bahn anwesend ist. (Es gilt der Stellzeitpunkt, nicht der Zeitpunkt, an dem die Anweisung eingeht.)
- Eine Bahn wird an ein totes Ende des Gleisnetzes geleitet.
- Eine Bahn erreicht eine Ausfahrt aus dem Gleisnetz, hat aber ihre verlangte Route nicht absolviert (wurde also fehlgeleitet).

7.8.7.3 Optionen

In diesem Abschnitt werden nun alle Menüpunkte detailliert beschrieben.

7.8.7.3.1 Main Menu – New Simulation Setzt eine Simulation in den Zustand vor der Konfiguration zurück. Alle Bahnen werden entfernt, alle Schaltelemente gehen in den Zustand `undefined` über. Um die Simulation zu starten, muss nun zunächst eine Konfiguration vorgenommen werden.

Diese Option ist nur verfügbar, wenn bereits ein Gleisnetz geladen wurde.
Die Tastenkombination für diese Option ist Ctrl-N.

7.8.7.3.2 Main Menu – Load TND Öffnet einen Dateiauswahldialog und lädt die ausgewählte Datei. Handelt es sich um eine korrekte TND, kann nun die Konfiguration der Simulation vorgenommen werden.

Diese Option bewirkt quasi das Gleiche wie `New Simulation`, nur dass noch ein Laden einer neuen TND vorgeschaltet ist.
Die Tastenkombination für diese Option ist Ctrl-L.

7.8.7.3.3 Main Menu – Quit Program Beendet den Simulator.

Die Tastenkombination für diese Option ist Ctrl-Q.

7.8.7.3.4 Options – Configure Random Simulation Leitet die Konfiguration einer Simulation ein, bei der nur die Anzahl der Bahnen gewählt werden soll. Es erscheint ein Dialog, in dem der Nutzer diese Zahl angeben kann. Die Bahnen werden sich aus den in der TND definierten Routen zufällig welche auswählen. Die Weichen und Signale bleiben im `undefined`-Zustand, bis sie erstmalig einen Schaltbefehl erhalten.

Wenn eine Simulation mit dieser Option konfiguriert wurde, bleibt sie auch während der laufenden Simulation verfügbar, so dass im Betrieb die Zahl der Bahnen verändert werden kann. Falls die Zahl verringert wird, wird allerdings keine Bahn unmittelbar aus dem Netz entfernt, sondern man muss abwarten, bis sie normal aus dem Netz gefahren ist.

Diese Option ist nur verfügbar, wenn bereits ein Gleisnetz geladen wurde.
Die Tastenkombination für diese Option ist Ctrl-R.

7.8.7.3.5 Options – Configure Preset Simulation Dies ist ebenfalls eine Möglichkeit, eine Simulation zu konfigurieren, jedoch um einiges umfassender als die vorherige Option. Nach der Auswahl der Bahnanzahl steht als nächstes die Festlegung von Einstellungen für die Bahnen an, diese sind:

- Route
  Die Route ist die Strecke, die die Straßenbahn entlang fahren soll. Es kann eine Route aus der Liste aller Routen ausgewählt werden. Nach Start der Simulation wird die Bahn an einer Einfahrt in das Gleisnetz eintreten, die zu einem RR-Sensor führt, der die gewünschte Route anfordern kann. (Dazu ist es wichtig, dass sich zwischen Einfahrt und RR-Sensor keine Weiche befindet, da sonst die Bahn nicht garantiert bis zum RR-Sensor gelangt, da für die Bahn kein Einfluss auf Stellung der Weiche besteht. Aus gleichem Grund darf kein Signal dazwischen vorhanden sein.) Hat die Bahn die Route erfolgreich absolviert und das Netz verlassen, wird sie kurz darauf wieder an bereits genannter Einfahrt erscheinen und ihre Route erneut absolvieren. Es handelt sich also hier um Bahnen, die ihre Routen immer wiederholt befahren.

- Länge
  Die Länge gibt an, wie lang die Straßenbahn sein soll. Es kann eine Länge aus einer Liste ausgewählt werden.

- maximale Geschwindigkeit
  Die maximale Geschwindigkeit gibt an, wie schnell diese Straßenbahn maximal fahren kann. Es kann eine Geschwindigkeit aus einer Liste ausgewählt werden.

- „May fail“
  Durch Aktivieren dieser Checkbox setzt der Nutzer die Option, dass eine Bahn während des Betriebes ausfallen kann. Sie bleibt dann irgendwann stehen, an welchem Ort dies geschieht, ist nicht vorhersehbar.

- „Random behavior“
  Eventuell könnte der Fahrer dieser Bahn sich zum Missachten von Signalen entschließen...

Es folgen die Einstellungen für Weichen:

- Stellung
  Die Stellung der Weiche, in der sie zu Beginn der gezielten Simulation sein wird. Aus einer Liste kann eine Stellung ausgewählt werden. Die Elemente dieser Liste sind abhängig von dem Typ der Weiche. So ist bei einer LR-Weiche nicht die Stellung `straight` möglich.

- „May fail“
  Durch Aktivieren dieser Checkbox setzt der Nutzer die Option, dass eine Weiche während des Betriebes ausfallen kann. Sie wird dann keine Schaltbefehle mehr ausführen. Wann der Ausfall eintritt, ist unvorhersehbar.

- „Random behavior“
  Die Weiche wird sich potentiell eigenmächtig umstellen, wenn diese Option aktiviert ist.

Für Sensoren kann eingestellt werden:

- „May fail“
  So eingestellte Sensoren werden irgendwann keine Meldungen mehr geben.

- „Random behavior“
  Diese Option bewirkt, dass ein Sensor ggf. Rückmeldungen gibt, wenn keine Bahn anwesend ist.

Abschließend für Signale:

- Stellung
  Die Stellung gibt an, was das Signal zu Beginn der Simulation anzeigen soll. Aus einer Liste kann die gewünschte Stellung ausgewählt werden. Die Elemente der Liste sind abhängig vom Typ des Signals, d. h. es kann nur eine Stellung ausgewählt werden, die das Signal auch annehmen kann.

- RR
  Mit bis zu drei Checkboxen, je nach Vorhandensein der RR-Anzeigen am Signal, können diese RR-Anzeigen initial ein- oder ausgeschaltet werden.

- „May fail“
  Durch Aktivieren dieser Checkbox setzt der Nutzer die Option, dass ein Signal während des Betriebes ausfallen kann. Es wird dann keine Schaltbefehle mehr ausführen. Wann der Ausfall eintritt, ist unvorhersehbar.

- „Random behavior“
  Das Signal wird sich potentiell eigenmächtig umstellen, wenn diese Option aktiviert ist.

Wurde eine Konfiguration dieser Art vorgenommen, kann danach keine Konfigurationsänderung mehr erfolgen. Erst nach Betätigen von `New Simulation` oder `Load TND` ist dies wieder möglich.

Diese Option ist nur verfügbar, wenn bereits ein Gleisnetz geladen wurde.
Die Tastenkombination für diese Option ist Ctrl-P.

7.8.7.3.6 Options – Start Simulation Startet die Simulation bzw. setzt sie nach einer Pause fort.

Diese Option ist nur verfügbar, wenn bereits ein Gleisnetz geladen und die Simulation konfiguriert wurde.
Die Tastenkombination für diese Option ist Ctrl-E.

7.8.7.3.7 Options – Stop Simulation Unterbricht die Simulation.

Die Tastenkombination für diese Option ist Ctrl-D.

7.8.7.3.8 Options – Create Logfile Das Aktivieren dieser Option bewirkt, dass ein Logfile geführt wird, in dem Daten zur laufenden Simulation abgelegt werden. Das Logfile wird in der Datei `sim.log` abgelegt. Diese kann maximal 10.000.000 Bytes größ werden. (Soll dieser Wert oder der Dateiname verändert werden, so sind die Werte für `logsize` bzw. `logname` in
`src/sim/Simulator.java` zu verändern.)

7.8.7.3.9 Options – Random Defects Ist diese Option aktiviert, können Hardware-Elemente oder Bahnen spontan in Defekt-Zustände übergehen. Das gilt auch für solche, die in einer etwaigen Detail-Konfiguration nicht als „verwundbar“ markiert wurden sind.

Zu beachten ist, dass die Defekterzeugung mit drei Ebenen arbeitet. Die erste ist „nicht defekt“, die zweite „potentiell defekt“ und die dritte„defekt“. Eine Aktivierung dieser Option bewirkt, dass Elemente von Ebene 1 auf Ebene 2 übergehen können. Sind sie erst einmal auf Ebene 2, können sie nicht wieder zurück, auch falls die Option wieder abgeschaltet wird. Auf Ebene 2 befindliche Elemente können in jedem Moment, unabhängig von dem Betätigen dieser Option, Defekte zeigen und damit auf Ebene 3 wechseln, auf der sie dann bis ans Ende der Simulation verbleiben.

7.8.7.3.10 Info – About SimTracs Diese Option gibt einen kurzen Info-Text aus.

Arne Stahlbock

7.8.8 Reflexion

In diesem abschließendem Kapitel soll rückblickend über die Erkenntnisse und Erfahrungen berichtet werden, die bei der Implementierung des Simulators gewonnen wurden.

Unser Rückblick befasst sich mit zwei getrennten Aspekten unserer Arbeit – zum einen mit den inhaltlich gewonnenen Erkenntnissen und zum anderen mit den Erfahrungen über Gruppenarbeit und die speziellen Schwierigkeiten in unserer Arbeitsgruppe.

7.8.8.1 Planungsfehler

Im Rahmen der gesamten Planung des Simulators wurden von der Simulatorgruppe viele Fehler gemacht. Diejenigen, die wir bis dato erkannt haben, beschreiben wir in diesem Unterkapitel. Erkannt haben wir folgende unserer Fehler:

- keine(n) Zeitplan / Arbeitspakete aufgestellt
- Anforderungen erst während der Implementierung aufgestellt
- Konzept erst während der Implementierung aufgestellt
- Benutzung von Compiler Konstruktions Werkzeugen nicht rechtzeitig überdacht

Im Folgenden gehen wir näher auf die einzelnen Aspekte ein und beschreiben, wie es dazu gekommen ist und was wir daraus gelernt haben.

7.8.8.1.1 Zeitplan / Arbeitspakete Die Simulatorgruppe hat es versäumt, sich zu Beginn ihrer Arbeit einen Zeitplan aufzustellen und Arbeitspakete einzuteilen.
Das Fehlen eines Zeitplans hat es für das Projektmanagement schwierig gemacht, die Fortschritte des Simulators zu verfolgen und die Einhaltung des Gesamtzeitplans von TRACS zu bestimmen. Zwar waren die Arbeitsergebnisse nach einem Semester durchaus zufriedenstellend, doch das war nicht unbedingt abzusehen und hätte auch schief gehen können.
Die Arbeitspakete wurden nicht von vornherein festgelegt – dennoch haben sie sich sehr schnell ergeben, da ein Großteil der Arbeit von Einzelpersonen zu Hause erledigt wurde und dafür die Arbeit irgendwie aufgeteilt werden musste. Die Gefahr hierbei war jedoch, dass Arbeitspakete vergessen werden. In der Tat, es wurden Arbeitspakete vergessen, die glücklicherweise jedoch weder unseren Zeitplan gefährdeten noch andere Arbeiten schwieriger gestalteten. Dies ist der Vorteil einer Arbeitsgruppe, die nur an wenigen Stellen mit dem Rest des Projektes verknüpft ist.
Ab dem zweiten Semester der Arbeit am Simulator (d.h. dem dritten Projektsemester insgesamt) wurde nach einem Zeitplan gearbeitet. Dieser konnte auch im Wesentlichen eingehalten werden. Die noch aufgetretenen leichten Abweichungen sind durch Einbindung des zu dem Zeitpunkt letzten verbliebenen Gruppenmitgliedes in andere Aktivitäten innerhalb des Projektes, genannt seien Weltmodell und TND-Änderungen, die zu den jeweiligen Zeitpunkten als wichtiger erschienen, auch sinnvoll zu begründen. Ohne diese zusätzlichen Dinge hätte der Zeitplan vermutlich gehalten werden können.
Nach dem dritten Projektsemester war der Simulator quasi fertiggestellt, nur die Anbindung an den SI konnte noch nicht getestet werden. Dies (und die Behebung der dabei entdeckten Fehler / Problemstellen) wurde nach Fertigstellung eines betriebsfähigen SI im vierten Semester durchgeführt, ohne dass hierfür noch einmal ein Zeitplan aufgestellt wurde, was aber auch nicht nötig erschien.

7.8.8.1.2 Anforderungen erst während der Implementierung aufgestellt
Wir haben versäumt rechtzeitig die Anforderungen an den Simulator und damit auch den Sinn und Zweck im Rahmen des Projektes TRACS zu spezifizieren. Dies hatte zur Folge, dass wir vieles nachträglich einbauen mussten und dass wir zum Teil auch Features implementiert haben, die gar nicht notwendig gewesen wären. Eine vorherige Spezifikation der Anforderungen hätte uns die Arbeit folglich erleichtert, da nachträgliche Änderungen immer mehr Zeit zum Einbau in Anspruch nehmen.
Das gesamte Projekt TRACS hat es verpasst sich um ein physikalisches Weltmodell, was der gesamten Arbeit als Grundlage dient, zu kümmern – dies wurde erst zum Ende des zweiten Semesters begonnen. Der Simulatorgruppe ist aufgefallen, dass sie sich während des Semesters ihr eigenes Weltmodell zugrunde gelegt hat, welches aber mit keiner anderen Gruppe abgesprochen war und welches auch nirgendwo genau dokumentiert wurde. Auch dies kostete die Simulatorgruppe Zeit es nachzuholen.

7.8.8.1.3 Konzept erst während der Implementierung aufgestellt Die Simulatorgruppe hat für einzelne Bereiche immer erst ein Konzept aufgestellt, bevor mit der Implementierung angefangen wurde. Doch es fehlte ein Gesamtkonzept für die Zusammenhänge der Bereiche. Die einzelnen Teilkonzepte wurden so flexibel gehalten, dass unser Vorgehen erfreulicherweise keine großen Probleme für uns aufgeworfen hat. Dennoch hätten wir mit einem Gesamtkonzept in der Tasche noch strukturierter und effektiver arbeiten können.

7.8.8.1.4 Benutzung von Compiler Konstruktions Werkzeugen nicht rechtzeitig überdacht Für den Einlesevorgang der TND benötigen wir einen Compiler. Diesen haben wir zuerst selbst programmiert (ohne Generierungs-Tools). Erst als bereits wenige Personenwochen an Zeit vergangen waren, fiel uns der Zeitaufwand solch eines Compilers auf und wir wechselten um und entschieden uns für die Benutzung so genannter Compiler Konstruktions Werkzeuge (Lexer- und Parser-Generatoren). Dies hätte von unserer Seite kaum vermieden werden können, da wir keine Erfahrung mit der Implementierung solcher Compiler hatten und den Arbeitsaufwand nicht abschätzen konnten.

7.8.8.2 Gruppeninterne Erkenntnisse

Wie gut die Arbeit innerhalb der Gruppe funktionierte, was gut und was weniger gut bei der Kommunikation lief, wird nun folgend dargestellt.

7.8.8.2.1 Gruppengröße Die Gruppengröße von vier Personen halten wir für eine angemessene Anzahl. Die Arbeit an den verschiedenen Stellen im Simulator erlaubte eine einfache Aufteilung und bei vier Personen können auch relativ leicht Entscheidungen getroffen und Kompromisse gemacht werden. Auch bei Ausfall eines Mitglieds (z. B. aufgrund einer Krankheit) lässt es sich mit drei Personen noch gut arbeiten.
Nach einem Semester jedoch musste die Simulatorgruppe einen Rückgang auf eine Person hinnehmen. Zwei vorige Mitglieder arbeiteten auf den Bachelor-Abschluss hin und verließen das Projekt plangemäß. Diese Situation war aber lange vorher bekannt und daher in der weiteren Zeitabschätzung auch so berücksichtigt. Kurz nach Beginn des dritten Semesters schied jedoch ein weiterer Teilnehmer aus, so dass die Gruppe aus nunmehr einer Person bestand. Im Nachhinein betrachtet war diese Situation jedoch nicht schädlich für das Vorankommen der Arbeit, auch wenn eine größere Gruppe sicherlich wünschenswerter gewesen wäre. Unter Betrachtung der Bedeutung des Simulators für das Gesamtprojekt konnte diese Gruppengröße aber akzeptiert werden.

7.8.8.2.2 Kommunikation Die Kommunikation in der Gruppe war zum großen Teil sehr gut. Bei den wöchentlichen Treffen gab es gute Diskussionen und der Emailverkehr in der restlichen Zeit der Woche war sehr rege, trug zu Entscheidungsfindungen

bei und half bei Zwischenfragen. Die Beantwortungszeit auf eine Frage in einer Email war meist sehr gering, so dass dies kaum Verzögerungen bei der Programmierung nach sich zog. (Anmerkung: Dies gilt natürlich nur für die Zeit, als die Gruppe noch aus vier Personen bestand.)

7.8.8.2.3 Leistung Die Mitglieder arbeiteten mit einer Ausnahme alle gleichwertig bei der weiteren Planung, Diskussion und Lösungsfindung mit. Zu Beginn des zweiten Semesters wurde noch nicht sehr viel pro Woche geschafft – hier hätte man noch mehr Leistung bringen können. Jedoch waren die Simulator-Gruppenmitglieder nicht sehr motiviert durch die insgesamt sehr geringe Motivation im Projekt. Aber nach ein paar Wochen (kann man passender weise als ‚Startschwierigkeiten' bezeichnen) wurde die Arbeitsleistung erhöht und das Arbeitstempo angezogen. Die Leistungssteigerung wurde zusätzlich durch die Tatsache verstärkt, dass zwei Mitglieder der Simulator-Gruppe am Ende des angesprochenen Semesters das Projekt mit dem Bachelor-Abschluss verlassen und eine möglichst gute Anschlussnote bekommen möchten.
Die weitere Arbeit des im Anschluss an das zweite Semester letzten verbliebenen Gruppenmitglieds mögen andere beurteilen, eine Eigenbewertung scheint hier nicht angebracht.

7.8.8.2.4 Schwierigkeiten Die Arbeitsgruppe Simulator muss eingestehen, dass sie es das ganze zweite Semester über nicht geschafft hat, eines der vier Mitglieder zu integrieren. Das langsame Arbeiten eines Mitgliedes wurde zuerst noch mit sprachlichen Schwierigkeiten begründet, doch es stellte sich schnell heraus, dass das nicht das einzige Problem war. Es wurde versucht beim Programmieren mit Java™Hilfestellung zu geben – dabei wurde offensichtlich, dass es auch an Grundlagen bei der Programmierung mit Java™fehlte. Bei den Gruppentreffen wurde sehr früh erkennbar, dass dieses Mitglied den Anschluss zu verlieren drohte und es wurde von den anderen versucht darauf hinzuweisen, dass es mehr tun müsse. Es wurde sich darum bemüht, auf das vierte Mitglied Rücksicht zu nehmen, doch irgendwann war es klar – es hatte den Anschluss an das Projekt verloren.
Die restlichen drei Mitglieder fühlten sich an einer Gabelung angekommen. Die eine Möglichkeit bestand in dem Versuch, das letzte Mitglied auf Biegen und Brechen zu integrieren und es zur Mitarbeit zu ‚zwingen' und dabei auch viele Hilfestellungen zu geben (was die Weiterentwicklung des Simulators mehr oder minder zum Stillstand gebracht und auch nur dann geholfen hätte, wenn das vierte Mitglied sich sehr engagiert hätte). Die Alternative war das Ignorieren des vierten Mitgliedes bei der Arbeitsverteilung (was für das Vorankommen des Simulators hilfreich, dabei aber besagtem Mitglied die Integration unmöglich gemacht hätte). Man entschied sich für die zweite Möglichkeit und arbeitete bis zum Ende des Semesters praktisch zu dritt.
Wir sehen ein, dass wir erst viel zu spät mit dem vierten Mitglied über die Hintergründe

geredet haben – eine frühere Diskussion darüber hätte vermutlich die Chance auf eine gute Zusammenarbeit zu viert deutlich erhöht. Die drei Mitglieder gingen davon aus, dass das vierte Mitglied sie bei Problemen ansprechen würde, doch das geschah nicht. Das vierte Mitglied ging (aus Sicht der anderen drei) davon aus, dass man ihm helfen würde, auch ohne dass es darum bittet – auch das geschah schon sehr früh nicht mehr. Aufgrund dieser nicht vereinbarenden Haltungen kam es sehr früh zum Bruch innerhalb der Gruppe und eine echte Problemlösung fand nicht statt.
Nach Beginn des dritten Semesters schied das besagte Gruppenmitglied dann relativ schnell aus dem Projekt aus. Einerseits ist es zu begrüßen, dass auf diese Weise die doch relativ ungewisse und folglich nicht förderliche Situation bezüglich der Frage, ob das Mitglied noch auf eine Weise in das Projekt bzw. die Gruppe integriert werden könnte, dass eine für das Projekt nützliche Leistung erbracht werden würde, geklärt wurde, andererseits muss man diese Art der „Lösung“ auch als Versagen der Gruppe bewerten.

Silvia Graumann, Niklas Polke, Arne Stahlbock

Kapitel 8

Sicherheitsnachweis

8.1 Überblick

Die wesentliche Aufgabe von Bahnsteuerungssystemen ist es, Katastrophen wie Kollisionen oder Entgleisungen zu vermeiden. Das Steuerungssystem muss daher die Signale und Weichen so schalten, dass solche Ereignisse nicht auftreten können.
Um die Sicherheit des Systems nachzuweisen, werden zunächst alle möglichen Bedrohungen (Hazards) identifiziert, die dazu führen können, dass die Sicherheitsanforderungen nicht erfüllt sind. Die ermittelten Hazards beruhen auf dem Weltmodell. Jeder Hazard kann dabei unterschiedliche Ursachen haben. Aus diesen Ursachen werden diejenigen ermittelt, die durch das System zu beeinflussen sind.
Um die Hazards bzw. die Ursachen eines möglichen Hazards systematisch darzustellen, wird eine Fehlerbaumanalyse (Fault Tree Analysis) durchgeführt. So wird nachgewiesen, dass der jeweilige Hazard nicht auftreten kann, wenn sichergestellt wird, dass die Ursachen nicht auftreten können. Diese Fehlerbaumanalyse wird im Abschnitt 8.2 angegeben. Eine Auswertung des Fehlerbaums erfolgt im Abschnitt 8.3 auf Seite 435.

Taffou Happi, Ruben Rothaupt

8.2 Fehlerbaumanalyse

Das Ergebnis der Fehlerbaumanalyse lässt sich in einer Baumstruktur darstellen. Auf diese Weise lässt sich erkennen, welche Ursachen der jeweilige Hazard haben kann.

- Der senkrechte Strich (|) steht für eine Oder-Verknüpfung der Knoten auf der gleichen Ebene.
- Das Plus (+) steht für eine Und-Verknüpfung der Knoten auf der gleiche Ebene.

Eine Und-Verknüpfung bedeutet, dass die übergeordnete Ursache auftritt, wenn alle der untergeordneten Ursachen auftreten.
Eine Oder-Verknüpfung bedeutet, dass die übergeordnete Ursache auftritt, wenn mindestens eine der untergeordneten Ursachen auftritt.
Nun soll der entwickelte Fehlerbaum vollständig angegeben werden. Diejenigen Punkte, bei denen eine Beeinflussung durch TRACS möglich ist, werden mit XXX markiert. Dadurch, dass man garantieren kann, dass keiner dieser Punkte auftritt, wird nachgewiesen, dass die Sicherheitsanforderungen des Gesamtsystems erfüllt sind.

```
Die Sicherheitsbedingungen des Gesamtsystems
(Strassenbahnsteuerungssystem) werden nicht erfuellt

| 1. Kollisionen zwischen Bahnen
   | 1.1 Auffahrunfälle
      | 1.1.1 Fehlverhalten des Fahrers (zu geringer Sicherheitsabstand)
      | 1.1.2 Sensor/Signalausfall oder -defekt
         + 1.1.2.1 Sensor/Signalausfall oder -defekt
         + 1.1.2.2 Nicht alle betroffenen Signale = HALT (XXX)
      | 1.1.3 Zu viele Bahnen auf einem Abschnitt
         | 1.1.3.1 Zeitgleich mehr Bahnen auf einem Gleisabschnitt als
                   maximal möglich
      | 1.1.4 Verkehrszeichenprobleme
      | 1.1.5 Bahn bleibt plötzlich stehen
         | 1.1.5.1 Stromausfall
            + 1.1.5.1.1 Stromausfall in Teilgleisnetz
            + 1.1.5.1.2 Nicht alle betroffenen Signale = HALT (XXX)
         | 1.1.5.2 defekte Bahn
      | 1.1.6 Steuerungssystem
         + 1.1.6.1 Steuerungssystemausfall (XXX)
            | 1.1.6.1.1 Stromausfall (XXX)
            | 1.1.6.1.2 Hardwaredefekt (XXX - Auswahl der Hardware)
            | 1.1.6.1.3 Softwareprobleme (z.B. Absturz, Deadlock) (XXX)
         + 1.1.6.2 Nicht alle betroffenen Signale = HALT (XXX)
   | 1.2 Bahnen in entgegengesetzter Fahrtrichtung auf demselben Gleisabschnitt
      | 1.2.1 Bahnen auf entsprechend führende Gleise geleitet
              (n(g1,g2); dst(g1) = g2 && dst(g2) = g1)
         | 1.2.1.1 Signale so geschaltet, dass Bahnen aufeinander zufahren
            | 1.2.1.1.1 Störung eines Signals oder des zu ihm führenden
                        Übertragungsweges
               | 1.2.1.1.1.1 Nichtbefolgen von Schaltanweisungen
                  | 1.2.1.1.1.1.1 Schaltanweisung wird ignoriert
                  | 1.2.1.1.1.1.2 Schaltanweisung gelangt nicht zu Signal
                     | 1.2.1.1.1.1.2.1 wegen Defekt am Steuersystem (XXX)
                     | 1.2.1.1.1.1.2.2 wegen Defekt auf Übertragungsweg
               | 1.2.1.1.1.2 Eigenmächtige Schaltungen des Signals
            | 1.2.1.1.2 Fehlverhalten des Steuersystems
               | 1.2.1.1.2.1 Fehler in Projektierungsdaten
                  | 1.2.1.1.2.1.1 Benutzer gibt falsche Daten ein (XXX)
```

```
            | 1.2.1.1.2.1.2 Netzgraph liefert TND, die nicht der
                            Zeichnung entspricht (XXX)
            | 1.2.1.1.2.1.3 Compiler liefert Binärdaten, die nicht
                            der Eingabe-TND entsprechen (XXX)
         | 1.2.1.1.2.2 Steuersystem gibt eine nicht den
                       Projektierungsdaten entsprechende Anweisung
                       oder unterlässt eine in den
                       Projektierungsdaten geforderte Anweisung
            | 1.2.1.1.2.2.1 Hardware-Fehler am Steuersystem (XXX)
            | 1.2.1.1.2.2.2 Treiber-Fehler (XXX)
            | 1.2.1.1.2.2.3 SI-Fehler (XXX)
            | 1.2.1.1.2.2.4 Fehler in Betriebssystem o. Systemumgebung (XXX)
         | 1.2.1.1.2.3 wegen falscher Sensordaten
            | 1.2.1.1.2.3.1 Sensor reagiert nicht auf Bahn
            | 1.2.1.1.2.3.2 Sensor reagiert falsch auf Bahn
         | 1.2.1.1.2.4 falsche Reaktion auf erkannten
                       Sensor/Signalausfall oder -defekt
            | 1.2.1.1.2.4.1 Sensor/Signalausfall oder -defekt
              + 1.2.1.1.2.4.1.1 Sensor/Signalausfall oder -defekt
              + 1.2.1.1.2.4.1.2 Steuersystem schaltet nicht alle
                                betroffenen Signale auf Stop (XXX)
| 1.2.1.2 Weichen so geschaltet, dass Bahnen aufeinander zufahren
   | 1.2.1.2.1 Störung einer Weiche oder des zu ihr
                führenden Übertragungsweges
      | 1.2.1.2.1.1 Nichtbefolgen von Schaltanweisungen
         | 1.2.1.2.1.1.1 Schaltanweisung wird ignoriert
         | 1.2.1.2.1.1.2 Schaltanweisung gelangt nicht zu Weiche
            | 1.2.1.2.1.1.2.1 wegen Defekt am Steuersystem (XXX)
            | 1.2.1.2.1.1.2.2 wegen Defekt auf Übertragungsweg
      | 1.2.1.2.1.2 Eigenmächtige Schaltungen der Weiche
   | 1.2.1.2.2 Fehlverhalten des Steuersystems
      | 1.2.1.2.2.1 Fehler in Projektierungsdaten
         | 1.2.1.2.2.1.1 Benutzer gibt falsche Daten ein (XXX)
         | 1.2.1.2.2.1.2 Netzgraph liefert TND, die nicht der
                         Zeichnung entspricht (XXX)
         | 1.2.1.2.2.1.3 Compiler liefert Binärdaten, die nicht
                         der Eingabe-TND entsprechen (XXX)
      | 1.2.1.2.2.2 Steuersystem gibt eine nicht den
                    Projektierungsdaten entsprechende Anweisung
                    oder unterlässt eine in den
                    Projektierungsdaten geforderte Anweisung
         | 1.2.1.2.2.2.1 Hardware-Fehler am Steuersystem (XXX)
         | 1.2.1.2.2.2.2 Treiber-Fehler (XXX)
         | 1.2.1.2.2.2.3 SI-Fehler (XXX)
         | 1.2.1.2.2.2.4 Fehler in Betriebssystem o. Systemumgebung (XXX)
      | 1.2.1.2.2.3 wegen falscher Sensordaten
         | 1.2.1.2.2.3.1 Sensor reagiert nicht auf Bahn
         | 1.2.1.2.2.3.2 Sensor reagiert falsch auf Bahn
      | 1.2.1.2.2.4 falsche Reaktion auf erkannten
```

```
                        Sensor/Signalausfall oder -defekt
                | 1.2.1.2.2.4.1 Sensor/Signalausfall oder -defekt
                   + 1.2.1.2.2.4.1.1 Sensor/Signalausfall oder -defekt
                   + 1.2.1.2.2.4.1.2 Steuersystem schaltet nicht alle
                                     betroffenen Signale auf Stop (XXX)
   | 1.2.2 Bahn überfährt Stop-Signal
      | 1.2.2.1 Fehlverhalten des Fahrers
      | 1.2.2.2 Bremsendefekt an der Bahn (kann nicht anhalten)
   | 1.2.3 unzureichende Signalaustattung des Gleisnetzes
           (Abschnitte nicht zu schützen, weil kein Signal vorhanden)
   | 1.2.4 Signale schlecht sichtbar
      | 1.2.4.1 witterungsbedingt (d.h. vorübergehend)
      | 1.2.4.2 Signale dauerhaft verdeckt
   | 1.2.5 Bahn(en) bleibt stehen
      | 1.2.5.1 Stromausfall
         + 1.2.5.1.1 Stromausfall in Teilgleisnetz
         + 1.2.5.1.2 Nicht alle betroffenen Signale = HALT (XXX)
         | 1.2.5.1.3 Fehlverhalten des Fahrers (zu geringer Sicherheitsabstand)
      | 1.2.5.2 defekte Bahnen
   | 1.2.6 Steuerungssystem
      + 1.2.6.1 Steuerungssystemausfall
         | 1.2.6.1.1 Stromausfall (XXX)
         | 1.2.6.1.2 Hardwaredefekt (XXX)
         | 1.2.6.1.3 Softwareprobleme (z.B. Absturz, Deadlock) (XXX)
      + 1.2.6.2 Nicht alle betroffenen Signale = HALT (XXX)
| 1.3 Bahnen stoßen in die Flanken
   | 1.3.1 Bahnen auf entsprechend führende Gleise geleitet
           (n(g,g1); n(g,g2); dst(g1) = g && dst(g2) = g) ||
           (crossing((g1,g2)(g3,g4)); dst(g1)= g2 && dst(g3)= g4)
      | 1.3.1.1 Signale so geschaltet, dass Flankenkollision möglich
         | 1.3.1.1.1 Störung eines Signals oder des zu ihm führenden
                     Übertragungsweges
            | 1.3.1.1.1.1 Nichtbefolgen von Schaltanweisungen
               | 1.3.1.1.1.1.1 Schaltanweisung wird ignoriert
               | 1.3.1.1.1.1.2 Schaltanweisung gelangt nicht zu Signal
                  | 1.3.1.1.1.1.2.1 wegen Defekt am Steuersystem (XXX)
                  | 1.3.1.1.1.1.2.2 wegen Defekt auf Übertragungsweg
            | 1.3.1.1.1.2 Eigenmächtige Schaltungen des Signals
         | 1.3.1.1.2 Fehlverhalten des Steuersystems
            | 1.3.1.1.2.1 Fehler in Projektierungsdaten
               | 1.3.1.1.2.1.1 Benutzer gibt falsche Daten ein (XXX)
               | 1.3.1.1.2.1.2 Netzgraph liefert TND, die nicht der
                                Zeichnung entspricht (XXX)
               | 1.3.1.1.2.1.3 Compiler liefert Binärdaten, die nicht
                                der Eingabe-TND entsprechen (XXX)
            | 1.3.1.1.2.2 Steuersystem gibt eine nicht den
                          Projektierungsdaten entsprechende Anweisung
                          oder unterlässt eine in den
                          Projektierungsdaten geforderte Anweisung
```

```
            | 1.3.1.1.2.2.1 Hardware-Fehler am Steuersystem (XXX)
            | 1.3.1.1.2.2.2 Treiber-Fehler (XXX)
            | 1.3.1.1.2.2.3 SI-Fehler (XXX)
            | 1.3.1.1.2.2.4 Fehler in Betriebssystem o. Systemumgebung (XXX)
         | 1.3.1.1.2.3 wegen falscher Sensordaten
            | 1.3.1.1.2.3.1 Sensor reagiert nicht auf Bahn
            | 1.3.1.1.2.3.2 Sensor reagiert falsch auf Bahn
         | 1.3.1.1.2.4 falsche Reaktion auf erkannten
                       Sensor/Signalausfall oder -defekt
            | 1.3.1.1.2.4.1 Sensor/Signalausfall oder -defekt
               + 1.3.1.1.2.4.1.1 Sensor/Signalausfall oder -defekt
               + 1.3.1.1.2.4.1.2 Steuersystem schaltet nicht alle
                                 betroffenen Signale auf Stop (XXX)
   | 1.3.1.2 Weichen so geschaltet, dass Flankenkollision möglich
      | 1.3.1.2.1 Störung einer Weiche oder des zu ihr
                  führenden Übertragungsweges
         | 1.3.1.2.1.1 Nichtbefolgen von Schaltanweisungen
            | 1.3.1.2.1.1.1 Schaltanweisung wird ignoriert
            | 1.3.1.2.1.1.2 Schaltanweisung gelangt nicht zu Weiche
               | 1.3.1.2.1.1.2.1 wegen Defekt am Steuersystem (XXX)
               | 1.3.1.2.1.1.2.2 wegen Defekt auf Übertragungsweg
         | 1.3.1.2.1.2 Eigenmächtige Schaltungen der Weiche
      | 1.3.1.2.2 Fehlverhalten des Steuersystems
         | 1.3.1.2.2.1 Fehler in Projektierungsdaten
            | 1.3.1.2.2.1.1 Benutzer gibt falsche Daten ein (XXX)
            | 1.3.1.2.2.1.2 Netzgraph liefert TND, die nicht der
                            Zeichnung entspricht (XXX)
            | 1.3.1.2.2.1.3 Compiler liefert Binärdaten, die nicht der
                            Eingabe-TND entsprechen (XXX)
         | 1.3.1.2.2.2 Steuersystem gibt eine nicht den
                       Projektierungsdaten entsprechende Anweisung
                       oder unterlässt eine in den
                       Projektierungsdaten geforderte Anweisung
            | 1.3.1.2.2.2.1 Hardware-Fehler am Steuersystem (XXX)
            | 1.3.1.2.2.2.2 Treiber-Fehler (XXX)
            | 1.3.1.2.2.2.3 SI-Fehler (XXX)
            | 1.3.1.2.2.2.4 Fehler in Betriebssystem o. Systemumgebung (XXX)
         | 1.3.1.2.2.3 wegen falscher Sensordaten
            | 1.3.1.2.2.3.1 Sensor reagiert nicht auf Bahn
            | 1.3.1.2.2.3.2 Sensor reagiert falsch auf Bahn
         | 1.3.1.2.2.4 falsche Reaktion auf erkannten
                       Sensor/Signalausfall oder -defekt
            | 1.3.1.2.2.4.1 Sensor/Signalausfall oder -defekt
               + 1.3.1.2.2.4.1.1 Sensor/Signalausfall oder -defekt
               + 1.3.1.2.2.4.1.2 Steuersystem schaltet nicht alle
                                 betroffenen Signale auf Stop (XXX)
| 1.3.2 Bahn überfährt Stop-Signal
   | 1.3.2.1 Fehlverhalten des Fahrers
   | 1.3.2.2 Bremsendefekt an der Bahn (kann nicht anhalten)
```

```
      | 1.3.3 unzureichende Signalaustattung des Gleisnetzes
             (Abschnitte nicht zu schützen, weil kein Signal vorhanden)
      | 1.3.4 Signale schlecht sichtbar
         | 1.3.4.1 witterungsbedingt (d.h. vorübergehend)
         | 1.3.4.2 Signale dauerhaft verdeckt
      | 1.3.5 Bahn(en) bleibt stehen
         | 1.3.5.1 Stromausfall
            + 1.3.5.1.1 Stromausfall in Teilgleisnetz
            + 1.3.5.1.2 Nicht alle betroffenen Signale = HALT (XXX)
            | 1.3.5.1.3 Fehlverhalten des Fahrers (zu geringer Sicherheitsabstand)
         | 1.3.5.2 defekte Bahnen
      | 1.3.6 Steuerungssystem
         + 1.3.6.1 Steuerungssystemausfall
            | 1.3.6.1.1 Stromausfall (XXX)
            | 1.3.6.1.2 Hardwaredefekt (XXX)
            | 1.3.6.1.3 Softwareprobleme (z.B. Absturz, Deadlock) (XXX)
         + 1.3.6.2 Nicht alle betroffenen Signale = HALT (XXX)

| 2. Entgleisungen
   | 2.1 zu hohe Geschwindigkeit
      | 2.1.1 Fehlverhalten des Fahrers
      | 2.1.2 fehlende oder unpassende Geschwindigkeitsbeschränkungen
      | 2.1.3 Nicht genuegend Abstand zwischen Signale/Verkehrzeichen und
              Weiche/Kurve/Kreuzung (bzgl. Bremsweg)
      | 2.1.4 Bremsendefekte
   | 2.2 Beschädigungen
      | 2.2.1 an Gleisen inkl. Weichen
         + 2.2.1.1 Beschaedigtes Gleis / beschaedigte Weiche
         + 2.2.1.2 Nicht alle betroffene Signal = HALT (XXX)
      | 2.2.2 an der Bahn
   | 2.3. Umschalten von Weichen während sie befahren werden
          (Weiche im Umschaltzustand && Signal != HALT) ||
          (Weiche im Umschaltzustand && ctr(g) != ctr(g1) + ctr(g2)))
      | 2.3.1 Anwesenheit von Bahnen wird dem Steuersystem nicht mitgeteilt
         | 2.3.1.1 unzureichende Sensorausstattung
         | 2.3.1.2 Sensordefekt
         | 2.3.1.3 Hardwaredefekt auf Übertragungsstrecke
      | 2.3.2 Anwesenheit von Bahnen wird dem Steuersystem mitgeteilt,
              aber von diesem nicht beachtet
         | 2.3.2.1 Hardwaredefekt am Steuersystem (XXX)
         | 2.3.2.2 Software setzt Sensormeldung nicht oder falsch um
            | 2.3.2.2.1 Treiber-Fehler (XXX)
            | 2.3.2.2.2 SI-Fehler (XXX)
            | 2.3.2.2.3 Fehler in Projektierungsdaten
               | 2.3.2.2.3.1 Benutzer gibt falsche Daten ein (XXX)
               | 2.3.2.2.3.2 Netzgraph liefert TND, die nicht
                             der Zeichnung entspricht (XXX)
               | 2.3.2.2.3.3 Compiler liefert Binärdaten, die nicht der
                             Eingabe-TND entsprechen (XXX)
```

```
      | 2.3.2.3 Fehler an Betriebssystem / Systemungebung (XXX)
| 2.4. Auffahren auf nicht auffahrbare Weichen
   | 2.4.1 Bahnen auf entsprechend führende Gleise geleitet
      | 2.4.1.1 Signale so geschaltet, dass Bahnen auf nicht
                auffahrbare Weichen zufahren
         | 2.4.1.1.1 Störung eines Signals oder des zu ihm
                     führenden Übertragungsweges
            | 2.4.1.1.1.1 Nichtbefolgen von Schaltanweisungen
               | 2.4.1.1.1.1.1 Schaltanweisung wird ignoriert
               | 2.4.1.1.1.1.2 Schaltanweisung gelangt nicht zu Signal
                  | 2.4.1.1.1.1.2.1 wegen Defekt am Steuersystem (XXX)
                  | 2.4.1.1.1.1.2.2 wegen Defekt auf Übertragungsweg
            | 2.4.1.1.1.2 Eigenmächtige Schaltungen des Signals
         | 2.4.1.1.2 Fehlverhalten des Steuersystems
            | 2.4.1.1.2.1 Fehler in Projektierungsdaten
               | 2.4.1.1.2.1.1 Benutzer gibt falsche Daten ein (XXX)
               | 2.4.1.1.2.1.2 Netzgraph liefert TND, die nicht der
                               Zeichnung entspricht (XXX)
               | 2.4.1.1.2.1.3 Compiler liefert Binärdaten, die nicht der
                               Eingabe-TND entsprechen (XXX)
            | 2.4.1.1.2.2 Steuersystem gibt eine nicht
                          den Projektierungsdaten entsprechende Anweisung
                          oder unterlässt eine in den
                          Projektierungsdaten geforderte Anweisung
               | 2.4.1.1.2.2.1 Hardware-Fehler am Steuersystem (XXX)
               | 2.4.1.1.2.2.2 Treiber-Fehler (XXX)
               | 2.4.1.1.2.2.3 SI-Fehler (XXX)
               | 2.4.1.1.2.2.4 Fehler in Betriebssystem o. Systemumgebung (XXX)
            | 2.4.1.1.2.3 wegen falscher Sensordaten
               | 2.4.1.1.2.3.1 Sensor reagiert nicht auf Bahn
               | 2.4.1.1.2.3.2 Sensor reagiert falsch auf Bahn
            | 2.4.1.1.2.4 falsche Reaktion auf erkannten
                          Sensor/Signalausfall oder -defekt
               | 2.4.1.1.2.4.1 Sensor/Signalausfall oder -defekt
                  + 2.4.1.1.2.4.1.1 Sensor/Signalausfall oder -defekt
                  + 2.4.1.1.2.4.1.2 Steuersystem schaltet nicht alle
                                    betroffenen Signale auf Stop (XXX)
      | 2.4.1.2 Weichen so geschaltet, dass Bahnen Bahnen auf nicht
                auffahrbare Weichen zufahren
         | 2.4.1.2.1 Störung einer Weiche oder des zu ihr
                     führenden Übertragungsweges
            | 2.4.1.2.1.1 Nichtbefolgen von Schaltanweisungen
               | 2.4.1.2.1.1.1 Schaltanweisung wird ignoriert
               | 2.4.1.2.1.1.2 Schaltanweisung gelangt nicht zu Weiche
                  | 2.4.1.2.1.1.2.1 wegen Defekt am Steuersystem (XXX)
                  | 2.4.1.2.1.1.2.2 wegen Defekt auf Übertragungsweg
            | 2.4.1.2.1.2 Eigenmächtige Schaltungen der Weiche
         | 2.4.1.2.2 Fehlverhalten des Steuersystems
            | 2.4.1.2.2.1 Fehler in Projektierungsdaten
```

```
                  | 2.4.1.2.2.1.1 Benutzer gibt falsche Daten ein (XXX)
                  | 2.4.1.2.2.1.2 Netzgraph liefert TND, die nicht der
                                  Zeichnung entspricht (XXX)
                  | 2.4.1.2.2.1.3 Compiler liefert Binärdaten, die nicht
                                  der Eingabe-TND entsprechen (XXX)
               | 2.4.1.2.2.2 Steuersystem gibt eine nicht den
                             Projektierungsdaten entsprechende Anweisung
                             oder unterlässt eine in den
                             Projektierungsdaten geforderte Anweisung
                  | 2.4.1.2.2.2.1 Hardware-Fehler am Steuersystem (XXX)
                  | 2.4.1.2.2.2.2 Treiber-Fehler (XXX)
                  | 2.4.1.2.2.2.3 SI-Fehler (XXX)
                  | 2.4.1.2.2.2.4 Fehler in Betriebssystem o. Systemumgebung (XXX)
               | 2.4.1.2.2.3 wegen falscher Sensordaten
                  | 2.4.1.2.2.3.1 Sensor reagiert nicht auf Bahn
                  | 2.4.1.2.2.3.2 Sensor reagiert falsch auf Bahn
               | 2.4.1.2.2.4 falsche Reaktion auf erkannten
                             Sensor/Signalausfall oder -defekt
                  | 2.4.1.2.2.4.1 Sensor/Signalausfall oder -defekt
                     + 2.4.1.2.2.4.1.1 Sensor/Signalausfall oder -defekt
                     + 2.4.1.2.2.4.1.2 Steuersystem schaltet nicht alle
                                       betroffenen Signale auf Stop (XXX)
      | 2.4.2 Bahn überfährt Stop-Signal
         | 2.4.2.1 Fehlverhalten des Fahrers
         | 2.4.2.2 Bremsendefekt an der Bahn (kann nicht anhalten)
      | 2.4.3 unzureichende Signalaustattung des Gleisnetzes
              (Abschnitte nicht zu schützen, weil kein Signal vorhanden)
      | 2.4.4 Signale schlecht sichtbar
         | 2.4.4.1 witterungsbedingt (d.h. vorübergehend)
         | 2.4.4.2 Signale dauerhaft verdeckt
   | 2.5 Hindernisse auf/nahe von Gleisen
   | 2.6 Planungs- und Baufehler

| 3. Kollisionen mit Hindernissen (nicht Bahnen)
   | 3.1 Hindernisse auf/nahe von Gleisen

| 4. Fahrgäste kommen zu Schaden, ohne dass Bahnen kollidieren
   | 4.1 Stürze
      | 4.1.1 Gebrechliche Fahrgäste
         | 4.1.2 Fahrgäste halten sich nicht fest
         | 4.1.3 stark ruckelige Fahrt
            | 4.1.3.1 zu ruckelige/starke Bremsvorgänge
               | 4.1.3.1.1 Bremsendefekte
               | 4.1.3.1.2 Signale schalten auf Stop,
                           wenn Bahn schon knapp davor (XXX)
               | 4.1.3.1.3 Fehlverhalten des Fahrers
            | 4.1.3.2 zu ruckeliges/starkes Beschleunigen
               | 4.1.3.2.1 Defekt an der Bahn
               | 4.1.3.2.2 Fehlverhalten des Fahrers
```

`| 5. Bahntechnik/Bahnen halten der Belastung nicht stand`

Arne Stahlbock, Andreas Kemnade, Taffou Happi, Ruben Rothaupt

8.3 Auswertung

Die Sicherheitsanforderungen sind nicht erfüllt, wenn eines der folgenden fünf Ereignisse auftritt:

- Kollisionen zwischen Bahnen
- Entgleisungen
- Kollisionen mit Hindernissen (nicht Bahnen)
- Fahrgäste kommen zu Schaden, ohne dass Bahnen kollidieren
- Bahntechnik/Bahnen halten der Belastung nicht stand

Von diesen Ereignissen kann das entwickelte Steuerungssystem Kollisionen zwischen Bahnen sowie Entgleisungen vermeiden.
Es sollen nun die Ursachen betrachtet werden, welche entweder zu Kollisionen zwischen Bahnen oder zu Entgleisungen führen können und bei denen eine Beeinflussung durch TRACS möglich ist.
Ein Fehlverhalten des Systems wäre in folgenden Fällen durch TRACS verschuldet:

- Fehler in Projektierungsdaten
- Steuerungssystem gibt eine nicht den Projektierungsdaten entsprechende Anweisung oder unterlässt eine in den Projektierungsdaten geforderte Anweisung
- falsche Reaktion auf erkannten Ausfall oder Defekt eines Hardwareelements
- Steuerungssystemausfall oder -defekt

Um einen Fehler in den Projektierungsdaten zu vermeiden, müssen die Ursachen ausgeschlossen werden, die zu fehlerhaften Projektierungsdaten führen können.

- Ob der Benutzer in den Verschlusstabellen falsche Daten eingegeben hat, wird durch Model Checking überprüft. Details dazu sind im Abschnitt 7.6 auf Seite 203 zu finden.

- Ob aus dem Netzgraph eine TND-Netzwerkbeschreibung erzeugt wird, die nicht der Zeichnung entspricht, wird überprüft, indem der Simulator aus der TND-Netzwerkbeschreibung eine Visualisierung des Gleisnetzes erstellt. Nun kann überprüft werden, ob diese Visualisierung und der Netzgraph auch ein identisches Gleisnetz wiedergeben. Details dazu, wie diese Visualisierung erstellt wird, sind im Abschnitt 7.8 auf Seite 313 zu finden.

- Ob der Compiler fälschlicherweise Binärdaten liefert, die zu Kollisionen oder Entgleisungen führen können und nicht der Eingabe-TND entsprechen, wird durch Model Checking überprüft. Details dazu sind im Abschnitt 7.6 auf Seite 203 zu finden.

Um zu vermeiden, dass das Steuerungssystem eine nicht den Projektierungsdaten entsprechende Anweisung gibt oder eine in den Projektierungsdaten geforderte Anweisung unterlässt, müssen die Ursachen, die dazu führen können, ausgeschlossen werden.
Ob die im Folgenden angegebenen Ursachen auftreten, wird durch ausführliche Tests überprüft. Details zu den Tests sind im Abschnitt 7.7 auf Seite 255 zu finden.

- Hardware-Fehler am Steuerungssystem

- Treiber-Fehler

- SI-Fehler

- Fehler in Betriebssystem oder Systemumgebung

Ob das Steuerungssystem bei einem erkannten Ausfall oder Defekt eines Hardwareelements alle betroffenen Signale auf STOP schaltet, wird ebenfalls durch Tests überprüft. Durch Tests wird auch überprüft, ob Defekte am Steuerungssystem beispielsweise durch Softwareprobleme wie Deadlocks oder durch Hardwaredefekte existieren.

Taffou Happi, Ruben Rothaupt

Kapitel 9

Bewertung

In diesem Kapitel soll das Projektergebnis und das Zustandekommen abschließend bewertet werden. Dabei sollen auch die aufgetretenen Probleme bei der Planung und der Zusammenarbeit beleuchtet und die wissenschaftliche Leistung herausgestellt werden.

9.1 Wissenschaftliche Leistung

Dieser Abschnitt soll eine Einschätzung über die im Projekt TRACS geleisteten Arbeiten in Bezug auf die Projektvorgaben geben. Hierbei dienten die Quellen [HP02], [HP03b], [HP03a], und [PGHD04] als Arbeitsbasis. Im Folgenden werden die zentralen Konzepte innerhalb dieser Quellen und deren Umsetzung im Projekt TRACS beleuchtet. Hierbei werden sowohl nicht umgesetzte Teile der Vorgaben, als auch im Rahmen der TRACS Arbeit zusätzlich oder anders umgesetzte Anforderungen an ein Gleisnetzsteuerungssystem aufgezeigt.

9.1.1 Domänenspezifische Beschreibungssprache

Die in [HP02] vorgeschlagene domänenspezifische Beschreibungssprache für Gleisnetze wurde im Projekt TRACS entwickelt. Hierbei stehen in [HP02] vier sogenannte *Interlocking Tables* im Vordergrund, die alle notwendigen Informationen enthalten, die zum sicheren Steuern aller in einem Gleisnetz definierten Routen wichtig sind. Diese Tabellen finden in der vom Projekt TRACS entwickelten Beschreibungssprache *Tram Network Description* (siehe auch 7.1) ihre Entsprechung. Zusätzlich werden in der *Tram Network Description* noch weitere Informationen angegeben, welche primär für den von TRACS entwickelten Gleisnetzsimulator relevant sind.

9.1.1.0.5 Route Definition Table In [HP02][S. 3] wird eine *Route Definition Table* vorgeschlagen, eine Tabelle, die pro Route vorgibt, welche Sensoren jede definierte Route überfahren muss. Durch diese Informationen ist der Verlauf jeder Route durch ein Gleisnetz für ein Steuerungssystem eindeutig definiert. Diese Tabelle findet in der *Tram Network Description* eine Entsprechung. Der `Route Definitions` Block der *Tram Network Description* bildet die selben Informationen in textueller Form ab. Dort werden pro Route Listen von Sensoren angegeben. Jede Zeile dieses Blockes bildet somit eine Zeile der in [HP02][S. 3] vorgegebenen *Route Definition Table*.

9.1.1.0.6 Route Conflict Table Die in [HP02][S. 4] vorgegebene *Route Conflict Table* beschreibt tabellarisch, welche Routen innerhalb eines Gleisnetzes niemals gleichzeitig befahren werden dürfen. Werden solche konfliktierenden Routen gleichzeitig befahren, so kann es zu Kollisionen durch gemeinsam genutzte Gleisabschnitte, sowie zu Entgleisungen durch verschieden zu schaltende Weichen kommen. Informationen darüber, welche Routen miteinander in Konflikt stehen, werden in der *Tram Network Description* durch die Blöcke `Conflicts` und `Point-Conflicts` spezifiziert. Auch hier wird pro Route eine Liste angegeben, welche die mit dieser Route in Konflikt stehenden Routen angibt. Zusammen enthalten diese Blöcke die selben Informationen wie die in [HP02][S. 4] vorgeschlagene *Route Conflict Table*.

9.1.1.0.7 Point Position Table Um pro Route Informationen darüber zu haben, wie Weichen für welche Route geschaltet sein müssen, wird in [HP02][S. 4] die sogenannte *Point Position Table* vorgeschlagen. In dieser Tabelle wird pro Route für jede der zu überfahrenden Weichen dieser Route eine Richtung vorgegeben. Auch diese Informationen finden in der *Tram Network Description* eine direkte Entsprechung. Der Block `Conditions` enthält pro Route eine Liste von Weichen mit zugehöriger Richtung für diese Route. Er bildet daher die selben Informationen wie die in [HP02][S. 4] vorgeschlagene *Point Position Table* in textueller Form ab.

9.1.1.0.8 Signal Setting Table Analog zu den pro Route benötigten Weichenstellungen sind Informationen darüber notwendig, welche Stellung Signale pro zu befahrender Route haben müssen. Diese Informationen werden in der *Tram Network Description* durch den Block `Clearances` angegeben. Er entspricht in seinen Informationen dadurch der in [HP02][S. 4] vorgeschlagenen *Signal Setting Table*.

9.1.1.0.9 Zusätzliche Angaben Zur Steuerung eines Gleisnetzes benötigte das Projekt TRACS mehr Informationen, als in [HP02] gefordert. Daher wurden der *Tram Network Description* weitere Blöcke hinzugefügt, um solche Informationen spezifizieren zu können.

Da zum Steuern einzelner Hardwareelemente eines Gleisnetzes noch zusätzlich zu den oben angegebenen Informationen echte Hardwareeigenschaften wie Schaltzeiten oder Rückfallrichtungen notwendig sind, können in der *Tram Network Description* innerhalb der Blöcke `Signal-Properties` und `Point-Properties` Hardwareeigenschaften pro Signal bzw. Weiche angegeben werden. Genauere Informationen hierzu sind 7.1 zu entnehmen.
Da innerhalb des Projekts TRACS keine realen Gleisnetze gesteuert werden konnten, wurde ein Gleisnetzsimulator entwickelt (siehe 7.8), der ebenfalls auf der *Tram Network Description* aufbaut. Daher enthält sie zusätzliche Informationen über Koordinaten einzelner Hardwareelemente sowie deren Anordnungen zueinander. Diese Daten sind notwendig, um nur anhand einer *Tram Network Description* ein Gleisnetz visualisieren zu können, das ansonsten in seiner Topographie nicht eindeutig beschrieben wäre. Hierzu wurden die Blöcke `Coordinates`, `Relations` und `Signal-Positions` integriert. Zusätzlich werden Doppelweichen in der *Tram Network Description* erlaubt, die im Block `Slipswitches` zu spezifizieren sind.
Um eine Zuordnung von Gleisnetzelementen zu Treibertypen vornehmen zu können, wurde ein weiterer Block `Hardwaremap` in die *Tram Network Description* aufgenommen. Hier werden einzelnen Hardwareklassen, die den gleichen Treiber nutzen, entsprechende Gleisnetzelemente zugeordnet.
Der Block `Definitions` der *Tram Network Description* enthält keine zusätzlichen Informationen zum Steuern eines Gleisnetzes, vereinfacht aber die Weiterverarbeitung einer solchen Gleisnetzbeschreibung, da ein dafür zu nutzender Compiler einfacher wird.

9.1.2 Wiederverwendbare Architektur des Steuerungssystems

Der TRACS Steuerinterpreter bildet die wiederverwendbare Steuerungkomponente eines TRACS Gleisnetzsteuerungssystems. Sein Aufbau entspricht der in [HP02][S. 4] vorgeschlagenen Architektur.
Die in [HP02][S. 4] vorgegebene Aufteilung in *Route Dispatcher*, *Route Controller*, *Safety Monitor*, und *Hardware Abstraction Layer* wurde umgesetzt. Dadurch kann pro Route ein einzelner *Route Controller* seine eigene Route freischalten, überwachen und wieder sperren. Anforderungen für Routen werden vom *Route Dispatcher* entgegengenommen und je nach Befahrungskonstellation des Gleisnetzes an den entsprechenden *Route Controller* weitergegeben. Alle Steuerimpulse werden dann innerhalb des *Safety Monitor* auf Sicherheit überprüft, bevor sie an den *Hardware Abstraction Layer* weitergegeben werden, welcher aus Steuerimpulsen konkrete Treiberaufrufe herleitet.
Die in [HP03b] vorgeschlagene Semantik eines solchen Gleisnetzsteuerungssystems wurde innerhalb des TRACS Steuerinterpreters umgesetzt. Aufgrund der Struktur von Projektierungsdaten für den TRACS Steuerinterpreter enthält der *Safety Monitor* allerdings innerhalb seiner Statechart-Spezifikation nur eine Transition in den Sicheren Zustand, da entgegen der Vorgaben aus [HP03b] Sicherheitsbedingungen nicht getrennt vorliegen,

sondern als eine einzelne Invariante aus den Projektierungsdaten ausgelesen werden. Wird diese Invariante verletzt, veranlasst der *Safety Monitor* einen Übergang des Gleisnetzes in einen sicheren Zustand. Diese Abweichung der Vorgaben aus [HP03b] stellt allerdings nur einen formalen, nicht aber einen semantischen Unterschied dar.
Die Steuersemantik des TRACS Steuerinterpreters weicht insofern von den Vorgaben aus [HP03b] ab, als dass zusätzliche Gleisnetzeigenschaften wie Doppelweichen und Rückfallweichen integriert werden mussten. Das in [HP03b] vorgeschlagene Zustandsbasierte Verhalten wurde beibehalten und an den entsprechenden Stellen ergänzt. Die genaue Steuerungssemantik des TRACS Steuerinterpreters ist aus dessen formaler Spezifikation in 7.5 ersichtlich.

9.1.3 Verifikation eines Steuerungssystems

Um die Sicherheit von Projektierungsdaten für einen TRACS Steuerinterpreter gewährleisten zu können, werden diese durch Modellvergleiche geprüft. Hierbei werden gemäß [PGHD04] Modelle für Gleisnetze, Bahnbewegungen und Steuersysteme erzeugt. Der Zustandsraum des Gesamtmodells kann dann durch Model Checking auf unsichere Zustände überprüft werden.
Innerhalb von zu überprüfenden Projektierungsdaten finden sich notwendige Informationen, um ein Modell des von diesen Projektierungsdaten zu steuernden Gleisnetzes zu erstellen. Zusammen mit der Spezifikation des TRACS Steuerinterpreters (siehe 7.5) kann aus den Projektierungsdaten ebenfalls ein Modell des konkreten Steuersystems gewonnen werden. Modellvorgaben für mögliche Bahnbewegungen gehen aus dem TRACS Weltmodell hervor, in dem entsprechende Voraussetzungen formalisiert wurden. Diese drei Teilmodelle ergeben ein Gesamtmodell eines Steuerungssystems mit seiner zu steuernden Umwelt. Das Gesamtmodell ergibt nun einen Zustandsraum, welcher in jedem Zustand den Sicherheitsbedingungen gemäß [PGHD04][S. 6-11] entsprechen muss. Ist dies nicht der Fall, können die eingelesenen Projektierungsdaten nicht als sicher angesehen werden.
Das Projekt TRACS entschied sich, anstatt des in [PGHD04] vorgeschlagenen Model Checkers *SystemC* den Model Checker *NuSMV* zu nutzen. Dies hat aber keinen Einfluss auf das generelle Vorgehen. Die Vorgaben aus [PGHD04] wurden nur in einem anderen Werkzeug realisiert. Informationen zum TRACS Modellvergleich finden sich in Abschnitt 7.6.

9.1.4 Automatisierter Test von Steuerungssystemen

Ein generiertes TRACS Steuerungssystem bestehend aus TRACS Steuerinterpreter und Projektierungsdaten für ein Gleisnetz kann mithilfe einer automatisch generierten Testsuite auf Sicherheit getestet werden. Hierbei diente [HP03a] als Vorgabe.

Die wiederverwendbaren Komponenten des TRACS Steuerinterpreters *Route Dispatcher*, *Route Controller*, *Safety Monitor* und *Hardware Abstraction Layer* werden hierbei durch Modultests auf spezifikationsgemäßes Verhalten überprüft. Diese Tests sind Gleisnetzunabhängig, gehören aber trotzdem zur Testsuite eines Gleisnetzes.
Diese Komponenten werden dann in Subsystemtests im Zusammenspiel mit ihren Projektierungsdaten auf korrektes Verhalten getestet. Hierzu liest ein Testfallgenerator die Projektierungsdaten des zu testenden Subsystems ein, und generiert Testfälle für vollständige Zustands- und Übergangsüberdeckung. Ein Testorakel bedient sich ebenfalls der Projektierungsdaten, um gemäß [HP03a] und der in [HP03b] gegebenen Steuerungssemantik zu entscheiden, ob sich das Subsystem sicher verhält.
Im Software-Integrationstest wird ein vollständiges Steuerungssystem auf Korrektheit geprüft. Hierbei dienen wieder die Projektierungsdaten als Basis für Testfälle und Testorakel.
Der in [HP03a] vorgeschlagene Hardware-Software-Integrationstest konnte nicht durchgeführt werden, da im Kontext des Projektes TRACS keine konkrete Hardware benutzt wurde, und daher die für diese Form von Test benötigten Eingabe- und Ausgabehardware des Testlings und Testsystems nicht zur Verfügung stand.
Details zum automatischen Test von Gleisnetzsteuerungssystemen finden sich in Abschnitt 7.7.

9.1.5 Eigene Erweiterungen

Zusätzlich zu den Projektvorgaben aus [HP02], [HP03b], [HP03a], und [PGHD04] wurden weitere Komponenten entwickelt, welche die Generierung von Gleisnetzsteuerungssystemen erleichtern sollen.
Um ein Gleisnetz nicht textuell beschreiben zu müssen, wurde eine Komponente entwickelt, die es ermöglicht, ein Gleisnetz mithilfe des CAD Programms *QCAD* graphisch zu spezifizieren. Diese Grafik kann dann automatisch in eine *Tram Network Description* gewandelt werden. Allerdings können auf diese Art nur topologische Informationen erzeugt werden. Eine graphische Eingabemaske ermöglicht es, eine Gleisnetzbeschreibung in Form einer *Tram Network Description* komfortabel zu bearbeiten und zu vervollständigen. Mehr Informationen zu diesem sogenannten *TND-Builder* finden sich in Abschnitt 7.3.
Da im Projekt TRACS keine realen Gleisnetze gesteuert werden konnten, wurde ein Gleisnetzsimulator entwickelt, der ein Gleisnetz visualisiert und vom TRACS Steuerinterpreter gesteuert werden kann. Details hierzu finden sich in Abschnitt 7.8.

9.2 Projektarbeit und Management

9.2.1 Überblick

Dieser Bereich des Berichtes befasst sich mit der Projektarbeit als solcher. Es soll gezeigt werden, wie die Projektaufgabe angegangen, wie die Arbeit organisiert wurde und welche Erfahrungen dabei gemacht wurden. Nicht fehlen sollen auch Einschätzungen, ob und wie man bestimmte Dinge hätte besser lösen können.

Wir teilen den folgenden Abschnitt thematisch wie folgt auf:

- Verwaltungsgruppen
  Für organisatorische Dinge sowie für nicht auf das Projektthema bezogene, aber auch über die Dauer des Projektes anstehende Arbeiten wurden Verwaltungsgruppen eingerichtet.
- Arbeitsgruppen
  In diesen Gruppen wurde die eigentliche Entwicklungsarbeit geleistet.
- Projektplenum
  Wöchentlich – in der vorlesungsfreien Zeit seltener – fand ein Projektplenum statt.
- Zeitplanung
  Für eine erfolgreiche Arbeit ist eine Zeitplanung unerlässlich.
- Einschätzung
  Zum Schluss darf eine Bewertung nicht fehlen.

9.2.2 Verwaltungsgruppen

Zum Beginn des Projektes wurden sogenannte Verwaltungsgruppen gegründet, so wie es bei den meisten Hauptstudiumsprojekten im Studiengang Informatik an der Universität Bremen üblich ist, da auch die Organisation selbst ein Teil des Projektes ist und somit in studentischer Hand liegen soll.
Es wurde gleich zu Beginn klargestellt, dass von Seiten der Projektbetreuung gewünscht wird, dass die Gruppen rotieren sollen und jeder Teilnehmer möglichst einmal überall - und vor allem beim Projekt-Management - mitgewirkt haben soll. Dass dies eigentlich kaum möglich war, wurde alleine schon an der Anzahl der Projektteilnehmer deutlich.
Die im Projekt *TRACS* vertretenen Verwaltungsgruppen sind die folgenden:

- Projektmanagement
- Projekttag-Organisation

- Web
- Doku
- Sysadmins
- Service und Technik

9.2.2.1 Projektmanagement

Die vermeintlich wichtigste Verwaltungsgruppe war jene für das Projektmanagement - kurz PM. Das PM war dafür verantwortlich für jedes Plenum eine Tagesordnung zu erstellen, welche die zu behandelnden Themen beinhaltete. Außerdem war es dafür verantwortlich, den allgemeinen Ablauf im Projekt zu koordinieren, bei Problemen zwischen Gruppen und Teilnehmern zu vermitteln, etc.. Es war quasi das Bindeglied zwischen Projektteilnehmern und Projektbetreuern.

9.2.2.2 Projekttag-Organisation

Eine nur in der ersten Hälfte des letzten Projektsemesters aktive Gruppe war jene Gruppe zur Organisation des Projekttages, welcher gegen Ende eines jeden Hauptstudiumprojektjahrgangs stattfindet. Sie musste den Projektstand und die Präsentation des Projektes auf dem Projekttag als solches planen und organisieren.

9.2.2.3 Web

Die Webgruppe war für die Einrichtung und Gestaltung einer das Projekt beschreibenden Webseite zuständig. Während des gesamten Projektes musste diese Seite immer wieder aktualisiert werden, da über einen mit einem Passwort geschützten Bereich auch die Protokolle der einzelnen Projektplena und auch die Präsentationsfolien von gehaltenen Vorträgen als Download verfügbar gemacht wurden.

9.2.2.4 Doku

Die sogenannte Dokugruppe war nicht nur für die Erstellung eines Rahmens für die Dokumentation (dieser und andere Berichte), sondern war auch für die Einrichtung eines CVS-Repositories zuständig, mit welchem fortan gearbeitet wurde.

9.2.2.5 Sysadmins

Für die Rechner in unserem Projektraum, der uns von der Universität und dem Veranstalter (Arbeitsgruppe Betriebssysteme und verteilte Systeme) zur Verfügung gestellt wurde, musste es auch jemanden geben, der die Geräte betreut und einrichtet. Dazu wurde die Admingruppe ins Leben gerufen.

9.2.2.6 Service und Technik

Die Service- und Technikgruppe ist eine Zusammenlegung der drei zuletzt genannten Gruppen im letzten Semester des Projektes, da zu diesem Zeitpunkt bereits mehr als die Hälfte der ursprünglichen Teilnehmer das Projekt verlassen hatten und ein gewisser Personalmangel bestand. Diese Gruppe kümmerte sich also fortan um die Bereiche Webseite, CVS, Dokumentation und Projektraumrechner.

9.2.3 Arbeitsgruppen

Da das Projekt letztlich aus mehreren Teilaufgaben bestand, wurden verschiedene Arbeitsgruppen eingerichtet, die jeweils ein bestimmtes Aufgabenfeld bearbeiten sollten. Einige Gruppen existierten dabei über die gesamte Projektdauer, andere nur für kürzere Zeiträume.
Die Gruppen waren im Einzelnen:

- DSL
- Steuerinterpreter
- HW/SW-Test
- Modelchecking
- BSAG
- Compiler
- Netzgraph
- Simulator
- EVP
- TND-Builder
- Weltmodell, FTA

Diese Gruppen werden nun näher beschrieben.

9.2.3.1 DSL

Die DSL-Gruppe hatte die Aufgabe, eine domänenspezifische Beschreibungssprache für die Darstellung von Gleisnetzen zu entwickeln. Diese Gruppe war nach einer Reihe von Vorträgen auch in der Lage, dies zu bewerkstelligen, und konnte bis zum Ende des ersten Projektsemesters eine erste Version der als TND bezeichneten Sprache vorstellen.
Nach dem ersten Semester wurde diese Gruppe aufgelöst und die Teilnehmer in andere Gruppen eingeteilt. Bei den auch in der Folge noch nötigen Anpassungen und Erweiterung dieser Sprache wurde auf ehemalige Gruppenmitglieder zurückgegriffen.

9.2.3.2 Steuerinterpreter

Diese Gruppe war dafür zuständig, das eigentliche Steuersystem, bzw. den generischen Teil desselben, herzustellen. Dieses generische Steuersystem wird dann in Verbindung mit konkreten Projektierungsdaten zu einem konkreten Steuersystem. Zunächst war auch vorgesehen, dieser Gruppe die Aufgabe zu geben, Hardware zur testweisen Ansteuerung durch das Steuersystem auszuwählen bzw. zusammenzubauen (Arbeitstitel „Elektronikbasteleien“), dieser Teil wurde dann aber mangels Zeit verworfen. Ebenso aufgegeben wurde der Ansatz, für das Steuersystem ein spezielles Echtzeitbetriebssystem zu ermitteln und einzusetzen.

9.2.3.3 Test

Diese Gruppe war dafür zuständig, das erzeugte Steuersystem mittels Tests auf Korrektheit zu überprüfen. Später wurde diese Gruppe in HWSW-Gruppe (HWSW steht für Hardware-Software-Integrationstest) umbenannt.

9.2.3.4 Modelchecker

Die Modelchecker-Gruppe beschäftigte sich mit der Verifikation des Systems. Sie überprüfte die Sicherheitseigenschaften für die zu steuernden Gleisnetze.

9.2.3.5 BSAG

Die BSAG-Gruppe war im ersten Semester einzig und allein damit beschäftigt, mit der Bremer Straßenbahn AG (BSAG) Kontakt aufzunehmen, um Informationen zu vorhandenen Straßenbahnsteuerungssytemen zu erlangen. Anschließend wurde sie aufgelöst.

9.2.3.6 Compiler

Mit Beginn des zweiten Semesters wurde die Compilergruppe ins Leben gerufen, welche die Aufgabe hatte, einen Compiler für die benötigten Projektierungsdaten zu entwerfen, der als Eingabe ein Dokument in TND-Form bekommt.

9.2.3.7 Netzgraph

Ein Nebenprodukt des Projektwochenendes in Nienburg gegen Ende des ersten Semesters war der Netzgraph. Bis dato war niemals die Rede davon, dass eine graphische Darstellung eines Gleisnetzes mit Konvertierungstools zur Erzeugung einer Gleisnetzbeschreibung in TND-Form benötigt wird.
Diese Gruppe wurde also gegen Ende des ersten Semesters erschaffen und zu Beginn des zweiten Semesters mit neuer Besetzung fortgeführt. Sie hatte die Aufgabe der Konvertierung von mittels einer CAD-Software erstellten Gleisnetzen in die TND-Darstellung.

9.2.3.8 Simulator

Die Simulatorgruppe war ebenfalls ein Ergebnis des Projektwochenendes in Nienburg. Sie wurde ebenfalls zum zweiten Semester ins Leben gerufen und konnte ihre Aufgabe, die Erstellung einer Software zur Simulation von Gleisnetzen, die dann letztlich durch den SI gesteuert werden sollten, innerhalb von zwei Semestern – bis auf kleine Nacharbeiten bei Vorliegen des SI – erledigen, so dass danach wieder Arbeitskraft für andere Bereiche frei wurde.

9.2.3.9 EVP

Die Gruppe zu Entwicklungs- und Verifikationsprozess hatte die Aufgabe, die zwei Prozesse (Entwicklung des Gesamtsystems und Verifikation desselben) zu systematisieren und somit eine wichtige Grundlage für die weitere Arbeit und letztlich den Erfolg des Projektes zu legen. Sie wurde im Laufe des zweiten Semesters eingerichtet.

9.2.3.10 TND-Builder

Die TND-Builder-Gruppe war ein Ein-Mann-Projekt, das zu Beginn des vierten Semesters entstand und sich der Aufgabe annahm, ein Eingabetool für Verschlusstabellen und deren Integration in die TND zu entwickeln.

9.2.3.11 Weltmodell, FTA

Für einzelne Teilbereiche gab es keine eigentlichen Gruppen. Diese Teile wurden von einigen Projektteilnehmern nebenher angefertigt, wobei der Hauptteil der Arbeit auf das späte zweite und frühe dritte Semester fiel.

9.2.4 Projektplenum und weitere Treffen aller Teilnehmer

In der Vorlesungszeit wurde ein wöchentliches Projektplenum abgehalten. Auch in der vorlesungsfreien Zeit gab es Plena, dann allerdings seltener, zumeist zweiwöchentlich.

Darüber hinaus wurden – wenn auch nicht häufig – weitere Treffen aller Teilnehmer außerhalb des gewöhnlichen Rahmens anberaumt.

9.2.4.1 Plenum

Das Plenum war als das einzige regelmäßige Treffen aller Teilnehmer von erheblicher Bedeutung. Hier wurden vor allem Themen besprochen, die für das Gesamtprojekt von Belang waren, seien es organisatorische Dinge oder konkrete Punkte in der aktuellen Arbeit. Auch sollte jede Arbeitsgruppe – seltener auch die Verwaltungsgruppen – hier regelmäßig Bericht über ihre Arbeitsfortschritte erstatten. Darüber hinaus wurden, wo es sich anbot bzw. nötig erschien, Vorträge einzelner Teilnehmer oder von Kleingruppen gehalten.
Das Plenum wurde jeweils von einem Vertreter des aktuellen Projektmanagements moderiert. Das PM erstellte auch die jeweilige Tagesordnung. Außerdem wurde von jedem Plenum ein Protokoll angefertigt; die Pflicht zu protokollieren wechselte dabei reihum über alle Projektteilnehmer.

9.2.4.2 Projektwochenende

Gegen Ende des ersten Semesters fand das einzige Projektwochenende des Projektes im Naturfreundehaus Nienburg statt. Hier sollten möglichst viele Teilnehmer des Projektes sich ein Wochenende lang intensiver mit der Materie befassen, als dies in gewohnter Umgebung möglich erschien.
Im Wesentlichen bestand dieses Wochenende aus mehreren Plena und dazwischen eingeschobenen Kleingruppentreffen, in denen an der konkreten Thematik gearbeitet wurde. Darüber hinaus sollte dieses Wochenende auch dem besseren „Zusammenwachsen“ des Projektes dienen.

9.2.4.3 TRACS-Tag

Unter dieser Bezeichnung lief ein Tag, an dem über das normale Plenum hinaus gemeinsam gearbeitet wurde. Er stellte im Prinzip eine Mischung zwischen normalem Plenum und vollem Projektwochenende dar.

9.2.4.4 Besuch der Ausstellung „Das Depot“

Im Vorfeld eines Besuches bei der BSAG fand auch ein Besuch in der Ausstellung „Das Depot“ statt, die vom Verein „Freunde der Bremer Straßenbahn“ eingerichtet wurde. Hier informierten sich die Projektteilnehmer vor allem über Straßenbahntechnik der Vergangenheit.

9.2.4.5 Besuch der Bremer Strassenbahn AG (BSAG)

Gegen Ende des ersten Semesters fand ein von der BSAG-Arbeitsgruppe organisierter Besuch bei der BSAG statt. Das Projekt stellte sich dort der BSAG vor und erfuhr im Gegenzug einiges über die bei der BSAG eingesetzten Systeme.

9.2.4.6 Besuch von Hanning & Kahl

Im dritten Semester wurde Kontakt zu der Firma Hanning & Kahl hergestellt, die sich auch als relativ interessiert an unserer Thematik erwies. Dies führte letztlich zu einem Besuch eines Vertreters dieser Firma an der Universität Bremen. Ähnlich wie bei dem BSAG-Besuch stellten beide Seiten ihre Arbeit genauer vor.

9.2.5 Zeitplanung

Die Zeitplanung ist ein wesentlicher Bestandteil eines Projektes. Sie soll den Teilnehmern eines Projektes helfen, das Ziel nicht aus den Augen zu verlieren - nämlich die gestellte Aufgabe innerhalb des vorgegebenen Zeitraums möglichst vollständig zu erfüllen. Sie soll auch gewährleisten, dass die Teilnehmer bei den einzelnen Teilaufgaben nicht den Überblick für den Gesamtkontext verlieren. Letztlich stellt sie also eine Planungshilfe für alle Beteiligten dar.
In einem solchen Zeitplan werden die einzelnen Arbeitspakete der Teilgruppen mit ihren Abhängigkeiten zu andern Paketen und der veranschlagten Dauer, bzw. dem geplanten Fertigstellungstermin dargestellt.
Ein sinnvoller Zeitplan besteht also aus einer spinnennetzartigen Verkettung der Pakete und einer zeitlichen Darstellung auf einer Art Zeitschiene.
Die ersten Versuche einer Zeitplanung wurden im Laufe des zweiten Projektsemesters unternommen. Hierfür wurde eine Zeitplangruppe einberufen, die aus jeweils einem Vertreter der Arbeitsgruppen bestand. Es wurde auch ein Zeitplan aufgestellt, der jedoch aus verschiedenen Gründen schnell wieder überholt war. Im Laufe der Zeit wurden neue Zeitplan-Revisionen erstellt, wieder überholt, neu erstellt usw. Die Planungsarbeit verlagerte sich dabei mehr und mehr von der ursprünglichen Gruppe auf einen einzelnen Projektteilnehmer, der naturgemäß allerdings auf die von den Gruppen zu liefernden Angaben angewiesen war.

9.2.6 Abschließende Bewertung

Im Nachhinein muss man feststellen, dass das Projekt zwar einige seiner Ziele erreicht hat, jedoch nicht alle.
Es existiert eine domänenspezifische Beschreibungssprache, es existieren in Form des TND-Builders, der CAD-Objektbibliothek und des auf dieser aufbauenden Netzgraph-Konverters Eingabetools für diese Sprache, es existiert ein Projektierungsdaten-Com-

piler, ein Steuerinterpreter, ein Simulator. Die Projektierungsdaten können mit Modelchecking-Techniken auf ihre Sicherheit geprüft werden, auch Tests wurden in gewissem Umfang durchgeführt.
Gänzlich aufgegeben wurden jedoch eine Beschäftigung mit Hardware (sowohl mit zu steuernder Hardware wie auch mit der des Steuersystems), der Themenbereich Fail-Stop und der Einsatz von Echtzeitbetriebssystemen für das Steuersystem.
Eine erfolgreichere Arbeit wurde durch diverse Faktoren gehemmt oder gar verhindert. Diese Faktoren sollen im Folgenden beschrieben werden.

9.2.6.1 Probleme

9.2.6.1.1 Arbeitshaltung Gelegentlich entstand der Eindruck, dass sich das Interesse von Teilnehmern vor allem auf das „eigene“ Themengebiet innerhalb des Projektes konzentrierte. Dies äußerte sich z.B. darin, dass in Plenumssitzungen nicht mitdiskutiert wurde, falls es um „fremde“ Themen ging oder dass die Protokolle nicht gelesen wurden. Einige Teilnehmer wiesen auch eine beträchtliche Fehlquote auf.

Bei einigen Teilnehmern machte sich sehr schnell eine Art von Unlust breit. So gab es bereits nach den ersten zwei Wochen das Problem, dass ein Teilnehmer einer Referatsgruppe keinerlei Beitrag leistete, nicht zu Terminen erschien und somit mehr oder weniger aus dem Projekt entfernt wurde.
Zwei weitere Teilnehmer verließen dann im Lauf des ersten Semesters freiwillig das Projekt.
Auch im weiteren Verlauf zeigte sich besonders im zweiten Semester, dass einige Teilnehmer den Anforderungen wohl nicht gewachsen zu sein schienen, da mehr und mehr die Mitarbeit ausblieb und auch bei leichten Dingen Verständnisprobleme auftraten. Diese Teilnehmer waren ebenfalls bald als Abgänge zu verzeichnen.

9.2.6.1.2 Ungünstige Gruppeneinrichtung Die ursprünglichen Arbeitsgruppen wurden auf Wunsch der Projektbetreuer und des in jenem Semester amtierenden Projektmanagements relativ zu Beginn des Projektes eingeteilt. Diese Einteilung gleich am Anfang des Projektes vorzunehmen hat sich im Nachhinein als fatale Fehlentscheidung herausgestellt. Die Gruppen konzentrierten sich fortan vor allem auf ihren Bereich, wobei der Blick für das Gesamtprojekt weitgehend unterging. Im Nachhinein betrachtet hätten eher andere Teilgruppen, wie z.B. für Weltmodell und EVP zu diesem Zeitpunkt eingerichtet werden müssen, da diese Gruppen die Grundlagen für alle weiteren folgenden Gruppen hätten legen können.
Bei dem beschrittenen Vorgehen wurden diese wichtigen Themen daher erst später – zu spät – behandelt.

9.2.6.1.3 Unsystematische Arbeitsweise In mehreren Gruppen wurde zum Teil relativ unkoordiniert und unsystematisch gearbeitet. Dies äußerte sich z.B. bei der Aufstellung des Zeitplanes (siehe weiter unten) oder darin, dass Spezifikationen nicht als nötig erachtet und vernachlässigt wurden. Zu Zeitpunkten, da andere Gruppen von den Ergebnissen einer so arbeitenden Gruppe abhängig waren, blieben die negativen Wirkungen nicht aus.

9.2.6.1.4 Inter-Gruppen-Kommunikation Die Kommunikation zwischen den unterschiedlichen Arbeitsgruppen verlief nicht in allen Fällen zufriedenstellend. Zwar gibt es auch positive Beispiele, doch in diversen Fällen hätte es wesentlich besser sein müssen.

9.2.6.1.5 Ein-Mann-Gruppen Unter den auf die Gruppenarbeit bezogenen Probleme am schlimmsten waren jene, welche durch Teilnehmer verursacht wurden, welche noch nicht einmal an der gruppeninternen Kommunikation teilnahmen und somit die Arbeit in jenen Gruppen quasi vollkommen zum Erliegen brachten.
Das erste Opfer war somit die Netzgraphgruppe, welche seit Mitte des zweiten Semesters nur als Ein-Mann-Gruppe auftrat.
Zur Hälfte des Projektes dezimierte sich dann die Anzahl der Teilnehmer auf weniger als die Hälfte der ursprünglichen Zahl, so dass nun auch die Gruppen Simulator und HWSW als Ein-Mann-Gruppen auftraten.
Dass in solchen Gruppen die Produktivität nicht gerade die beste ist, ist verständlich, zumal dies auf die Motivation aller Teilnehmer drückt. Bei der Simulator-Gruppe war das Problem allerdings nur bedingt vorhanden, da schon lange vor Projekthalbzeit klar war, dass zwei Mitglieder das Projekt verlassen würden, da beide nicht wie der Rest der Teilnehmer das Diplom als angepeilten Abschluss anstrebten, sondern den Bachelor-Abschluss als Ziel hatten, welcher nur ein zwei-semestriges Projekt vorsah.
Alle anderen Gruppen hatten aber mit größeren Problemen zu kämpfen, da sämtliche Planungen auf mehr Teilnehmer ausgelegt waren und somit ein massives Problem mit der Zeitplanung auftrat.
Weiterhin kann in diesem Zusammenhang problematisch sein, dass bei Einzelpersonen die Ideen zur Lösung der Aufgaben relativ schnell erschöpft sein können und dass vorhandene Ideen von niemandem hinterfragt / kritisch betrachtet werden, womit die Gefahr, einen falschen Weg einzuschlagen, um einiges höher ist als bei mehreren Personen.

9.2.6.1.6 Konsummentalität In einigen Fällen muss festgestellt werden, dass Projektteilnehmer eine zu starke Konsum-Haltung an den Tag legten, ihnen also im Prinzip immer erst gesagt werden musste, was zu tun sei. Eigene Ideen wurden nicht oder nur in geringem Maße entwickelt. Außerdem kann eine solche Haltung dazu führen, dass derart eingestellte Teilnehmer die ihnen vorgelegten Konzepte und Ideen mehr oder weniger

kritiklos hinnehmen, da sie froh sind, überhaupt Ansatzpunkte zur Arbeit zu haben.

9.2.6.1.7 Art und Weise der Kritik Durch die Art und Weise, in der zuweilen Kritik vorgebracht wurde, erwies sich diese Kritik nicht immer als förderlich.
Beispiel: Wenn auf der einen Seite (zu Recht, siehe den vorigen Punkt) kritisiert wird, dass der eine oder andere Teilnehmer eine Konsummentalität an den Tag legt, aber auf der anderen Seite von den Teilnehmern eingebrachte Lösungswege als schlecht hingestellt werden, weil sie nicht dem Stand der Kunst entsprächen, dann muss man sich fragen, was eigentlich überhaupt gefordert wird. So etwas ist jedenfalls die Motivationsbremse schlechthin und führt dazu, dass ein Projekt zum Stillstand kommt. Nicht jede Standard-Methode ist immer die beste Methode um ein Problem zu lösen.

9.2.6.1.8 Zu spät aufgestellter Zeitplan Eine Zeitplanung wurde erst im Laufe des zweiten Semesters begonnen. Ein wesentlich besserer Zeitpunkt wäre irgendwann im ersten Semester gewesen. Eventuell wäre dies deutlicher geworden, hätte man zu Anfang die Grundlagen in Form eines EVP gelegt.

9.2.6.1.9 Mangelndes Interesse am Zeitplan Ein wesentliches Problem bei der Zeitplanung war das mangelnde Interesse an einem solchen Plan seitens eines gewissen Teils der Teilnehmer.
Es wurden von den entsprechenden Arbeitsgruppen entweder gar keine oder ungenaue Angaben zu den Arbeitspaketen gemacht, so dass nahezu jeder Zeitplan bei der Veröffentlichung eigentlich schon wieder hoffnungslos überarbeitungsbedürftig war.
Interessant auch die unverständliche Einstellung einiger Gruppen, dass man die Gruppenstärke nicht auf die veranschlagte Zeit umrechnen könne. Damit waren konsistente Zeitangaben schlicht unmöglich.

9.2.6.1.10 Probleme beim Definieren der Arbeitpakete Ein anderes Problem trat bei einigen Gruppen auf, die ihre Arbeitspakete nicht definieren konnten. So wurden für Kleinigkeiten mehrere Wochen angesetzt, während größere Teile für einen kurzen Zeitraum geplant waren. Oder es wurden Dinge in mehrere Pakete aufgesplittet, welche eigentlich zusammengehörten. Auch wurde einfach nur ein riesiges nicht näher definiertes Paket genannt, obwohl es in mehrere Teile aufsplittbar gewesen wäre.
All dies machte eine Zeitplanung zeitweilig extrem schwierig, da hierdurch auch viele Inkonsistenzen entstanden.

9.2.6.1.11 Fehleinschätzungen des Zeitaufwandes Ebenso passierte es hin und wieder, dass die eine oder andere Gruppe mit ihren Zeitschätzungen deutlich daneben lag, so dass die Fertigstellung eines Teils der Arbeit schon einmmal fast sechs Monate länger dauerte, wovon andere Gruppen mehr oder minder abhängig waren. Zu allem

Übel kam dann noch der vorzeitige Abgang diverser Projektteilnehmer dazu, so dass durch diesen Personalschwund sämtliche Zeitschätzungen revidiert werden mussten.

9.2.6.1.12 Unklare Betreuungssituation Es war wenig hilfreich, dass der eigentlich als Hauptbetreuer vorgesehene Professor gleich zu Beginn ein Forschungssemester einlegte und erst danach zum Projekt dazustoßen wollte, was letztlich niemals geschah. Während des ersten Semesters bestand so immer eine gewisse Unsicherheit – möglicherweise auch bei den Betreuern – die erst mit der endgültigen Absage aufgehoben wurde.

9.2.6.1.13 Begleitende Lehrveranstaltungen Auch war es alles andere als ideal, dass wesentliche begleitende Lehrveranstaltungen erst im zweiten oder gar dritten Projektsemester angeboten wurden, obwohl sie im Nachhinein betrachtet im ersten Semester hätten stattfinden müssen.

9.2.6.2 Fazit

Trotz der beschriebenen Probleme soll hier jedoch kein ausschließlich negatives Fazit gezogen werden. Gewisse, oben bereits genannte Ziele wurden erreicht, auch die Darstellung am Projekttag scheint auf Zustimmung des Publikums gestoßen zu sein. Einige Wochen vorher war wesentlich Schlimmeres zu befürchten.

Henrik Röhrup, Arne Stahlbock

Anhang A

TND-Grammatik

A.1 Lexergrammatik der TND, Version 3.0, 29. 4. 2005

```
"rr-sensors"          --    RRSENSORS
"td-sensors"          --    TDSENSORS
"tg-sensors"          --    TGSENSORS
"sd-sensors"          --    SDSENSORS
"sg-sensors"          --    SGSENSORS

"sl-points"           --    SLPOINTS
"sr-points"           --    SRPOINTS
"lr-points"           --    LRPOINTS
"slr-points"          --    SLRPOINTS

"l-signals"           --    LSIGNALS
"s-signals"           --    SSIGNALS
"r-signals"           --    RSIGNALS
"sl-signals"          --    SLSIGNALS
"sr-signals"          --    SRSIGNALS
"lr-signals"          --    LRSIGNALS
"slr-signals"         --    SLRSIGNALS

"routes"              --    ROUTES
"marks"               --    MARKS
"crossings"           --    CROSSINGS
"hwclasses"           --    HWCLASSES
```

```
"straight"              --    STRAIGHT
"left"                  --    LEFT
"right"                 --    RIGHT

"request-at"            --    REQUESTAT

"passive"               --    PASSIVE
"fallback"              --    FALLBACK
"breakable"             --    BREAKABLE
"wait-left"             --    WAIT_LEFT
"wait-right"            --    WAIT_RIGHT
"wait-straight"         --    WAIT_STRAIGHT
"rr-left"               --    RR_LEFT
"rr-right"              --    RR_RIGHT
"rr-straight"           --    RR_STRAIGHT
"switchtime"            --    SWITCHTIME

"definitions"           --    DEFINITIONS
"signal-properties"     --    SIGNAL_PROPERTIES
"point-properties"      --    POINT_PROPERTIES
"slip-switches"         --    SLIP_SWITCHES
"relations"             --    RELATIONS
"cs-relations"          --    CSRELATIONS
"ps-relations"          --    PSRELATIONS
"signal-positions"      --    SIGPOSITIONS
"routedefinitions"      --    ROUTEDEFINITIONS
"conditions"            --    CONDITIONS
"clearances"            --    CLEARANCES
"point-conflicts"       --    POINT_CONFLICTS
"conflicts"             --    CONFLICTS
"hardwaremap"           --    HARDWAREMAP
"coordinates"           --    COORDINATES

"entrance"              --    ENTRANCE
"exit"                  --    EXIT
"none"                  --    NONE

":" | ";" | "," | "\{"
| "\}" | "(" | ")"
| "-"                   --    yytext
```

```
"g"[[:alnum:]]+          --    SENSID
"w"[[:alnum:]]+          --    POINTID
"s"[[:alnum:]]+          --    SIGID
"r"[[:alnum:]]+          --    ROUTEID
"x"[[:alnum:]]+          --    MARKID
"c"[[:alnum:]]+          --    HWID
"k"[[:alnum:]]+          --    CROSSID

"A" | "B"                --    SIDE
"C" | "D"                --    CROSS_SIDE
"Y" | "L" | "S" | "R"    --    POINT_SIDE

-?[[:digit:]]+           --    COORD
[[:digit:]]+             --    POSINT

<*>[ \t\n]               --    /* Ignoriere sonstigen Whitespace. */
.                        --    ILLEGALTOKEN
```

A.2 Parsergrammatik der TND, Version 3.0, 29. 4. 2005

```
<tnd>          ::= <defblock> <coordblock> <sigpropblock>
                   [<ptpropblock>] [<slipswitchblock>]
                   <relblock> <sigposblock>
                   <routeblock> <condblock> <clearblock>
                   [<ptconfblock>] [<confblock>] <hwblock>

<defblock>     ::= <DEFINITIONS> "{" {<typdef>} "}"
<typdef>       ::= <sensdef> | <pointdef> | <sigdef> |
                   <routedef> | <markdef> | <hwdef> |
                   <crossdef>

<sensdef>      ::= <senstype> ":" <SENSID> {"," <SENSID>} ";"
<senstype>     ::= <RRSENSORS> | <TDSENSORS> | <TGSENSORS> |
                   <SDSENSORS> | <SGSENSORS>

<pointdef>     ::= <pointtype> ":" <POINTID> {"," <POINTID>} ";"
<pointtype>    ::= <SLPOINTS> | <SRPOINTS> | <LRPOINTS> |
                   <SLRPOINTS>

<sigdef>       ::= <sigtype> ":" <SIGID> {"," <SIGID>} ";"
<sigtype>      ::= <LSIGNALS> | <SSIGNALS> | <RSIGNALS> |
                   <SLSIGNALS> | <SRSIGNALS> | <LRSIGNALS> |
                   <SLRSIGNALS>

<routedef>     ::= <ROUTES> ":" <ROUTEID> {"," <ROUTEID>} ";"

<markdef>      ::= <MARKS> ":" <MARKID> {"," <MARKID>} ";"

<hwdef>        ::= <HWCLASSES> ":" <HWID> {"," <HWID>} ";"

<crossdef>     ::= <CROSSINGS> ":" <CROSSID> {"," <CROSSID>} ";"

<sigpropblock> ::= <SIGNAL_PROPERTIES> "{" {<sigpropdef>} "}"
<sigpropdef>   ::= <SIGID> ":" [<WAIT_STRAIGHT>] [<WAIT_LEFT>]
                   [<WAIT_RIGHT>] [<RR_STRAIGHT>] [<RR_LEFT>]
                   [<RR_RIGHT>] <SWITCHTIME> <POSINT> ";"
```

```
<ptpropblock>  ::= <POINT_PROPERTIES> "{" {<ptpropdef>} "}"
<ptpropdef>    ::= <POINTID> ":" [<PASSIVE>] [<FALLBACK> <direction>]
                   [<BREAKABLE>] <SWITCHTIME> <POSINT> ";"

<slipswitchblock> ::= <SLIP_SWITCHES> "{" {<slipswitchdef>} "}"
<slipswitchdef>   ::= <CROSSID> ":" <POINTID> "," <POINTID>
                      ["," <POINTID> "," <POINTID>]
                      ":" <SWITCHTIME> <POSINT> ";"


<relblock>       ::= <RELATIONS> "{" {<reldef>} "}"
<reldef>         ::= <relation> | <relnone>
<relation>       ::= <elementid-side> ":" {<MARKID> ","}
                     <elementid-side> ";"
<relnone>        ::= <sensid-side> ":" {<MARKID> ","} <een> ";"
<een>            ::= <ENTRANCE> | <EXIT> | <NONE>
<elementid-side> ::= <sensid-side> | <crossid-side> | <pointid-side>
<sensid-side>    ::= <SENSID> <SIDE>
<crossid-side>   ::= <CROSSID> <cross-side>
<cross-side>     ::= <SIDE> | <CROSS_SIDE>
<pointid-side>   ::= <POINTID> <POINT_SIDE>

<sigposblock>  ::= <SIGPOSITIONS> "{" {<sigposdef>} "}"
<sigposdef>    ::= <SIGID> ":" [<sensid-side> "-" <MARKID> "-"]
                   <sensid-side> ";"

<routeblock>   ::= <ROUTEDEFINITIONS> "{" {<rtdef>} "}"
<rtdef>        ::= <ROUTEID> ":" <SENSID> "," <SENSID> {"," <SENSID>}
                   "-" <REQUESTAT> ":" <SENSID> {"," <SENSID>} ";"

<condblock>    ::= <CONDITIONS> "{" {<conddef>} "}"
<conddef>      ::= <ROUTEID> ":" <condition>
                   {"," <condition>} ";"
<condition>    ::= <POINTID> <direction>
<direction>    ::= <LEFT> | <STRAIGHT> | <RIGHT>

<clearblock>   ::= <CLEARANCES> "{" {<cleardef>} "}"
<cleardef>     ::= <ROUTEID> ":" <clearance>
                   {"," <clearance>} ";"
<clearance>    ::= <SIGID> <direction>
```

```
<ptconfblock>  ::= <POINT_CONFLICTS> "{" {<confdef>} "}"
<confdef>      ::= <ROUTEID> ":" <ROUTEID> {"," <ROUTEID>} ";"

<confblock>    ::= <CONFLICTS> "{" {<confdef>} "}"

<hwblock>      ::= <HARDWAREMAP> "{" {<hwdefs>} "}"
<hwdefs>       ::= <HWID> ":" <hwelid> {"," <hwelid} ";"
<hwelid>       ::= <SENSID> | <POINTID> | <SIGID> | <CROSSID>

<coordblock>   ::= <COORDINATES> "{" {<coorddef>} "}"
<coorddef>     ::= <elementid> ":" "(" <COORD> ","
                   <COORD> "," <COORD> ")" ";"
<elementid>    ::= <SENSID> | <POINTID> | <SIGID> | <MARKID> | <CROSSID>
```

Anhang B

DXF-Datei-Format

B.1 Aufbau einer DXF-Datei

In einer DXF-Datei gibt es zwei Typen von Zeilen:

- eine Zeile, welche einen Code liefert, welcher als eine Art TAG anzusehen ist (der Code ist 1-3stellig und ggf. um 1-2 Blanks eingerückt (rechtsbündig))
- eine Zeile, die direkt im Anschluss an die erste folgt und den Wert des TAGS beinhaltet

Diese Zeilenpaare müssen immer als eine Einheit gesehen werden.

B.1.1 Für TRACS relevante Codes

Übersicht über die für den Netzgraphen relevanten DXF-Codes: (der Rest kann getrost ignoriert werden)

```
999:      Dateianfang
0:        bezeichnet irgendwelche Steuerabschnitte, Objekttypen, etc.
          (SECTION / ENDSEC / ENTITIES / BLOCKS / TABLES / LINE / CIRCLE
          / POINT / TEXT / EOF / ...)
8:        Name des Layers (bei uns: 'signals', 'sensors', 'track' oder
          'others')
10,20,30: bezeichnen im Allgemeinen Startpunkt (Koordinaten X,Y,Z) oder
          Mittelpunkt oder INSERT-Point (abh. vom Typ des Objektes)
11,21,31: wie oben, nur diesmal Endpunkt
40:       Radius (CIRCLE) oder Text-Höhe (TEXT)
41;       Text-Breite (width) (TEXT)
```

```
50:       Start-Winkel (ARC) oder Text-Rotation oder INSERT-Rotation
51:       End-Winkel (ARC)
1:        Text-Value
7:        Text-Style-Name (def=STANDARD)
71:       Text-Geneartion-Flags (opt.)
72:       Horizontal Text Justification Type (def=0)
73:       Fixed-Length
39:       Thickness (Text)
2:        INSERT-Blockname // Blockname vor Flags/Koords
          (in BLOCK-SECTION)
3:        Blockname nach Flags/Koords (in BLOCK-SECTION)
41,42,43: X/Y/Z-Scale-Factors (def=1)
```

B.1.2 Erläuterungen

In unserem Fall wird es außer dem Entity INSERT vermutlich nur noch ein LINE in der ENTITIES-SECTION geben. Ggf. können noch Text-Blöcke eingefügt werden, welche aber nicht weiter ausgelesen werden. Der Rest wird in der BLOCK-SECTION definiert. Hier gibt es nur LINE, CIRCLE, TEXT und INSERT als Entities.

Zu jedem Entity gehört auf jeden Fall eine X/Y/Z-Koordinate. Dies gilt auch für alle INSERTs innerhalb der BLOCKS-SECTION. Bei LINE kommt auch noch der 2. Koordinatensatz (11,21,31) als Endpunkt hinzu. Ebenfalls ist immer das Layer (8) definiert. Der Code 40 ist sowohl bei TEXT als auch bei CIRCLE zu finden. Bei TEXT gibt es noch die Textbreite (41) und natürlich den Wert (1). Der Drehwinkel ist bei TEXT und INSERT-Entities i.d.R. auch vorhanden (50). Weitere Codes oder Tags sind teilweise auch noch vorhanden oder sie sind optional. Näheres ist der DXF-Reference [Aut03] zu entnehmen.

In sämtlichen bekannten/untersuchten Versionen des DXF-Formats hat sich an der eigentlichen Syntax nichts geändert. Die Versionen sind abwärtskompatibel.
Wir verwenden keine der neueren Codes und brechen die DXF-Datei sozusagen auf ein rudimentäres Format herunter, in dem keinerlei der neueren Verwaltungsinformationen vorkommen. Mit QCAD ist eine solche vereinfachte DXF-Datei darstellbar. Sollten andere CAD-Programme dies nicht können, so muss eben QCAD verwendet werden.

Henrik Röhrup

B.2 Vereinfachte DXF-Grammatik

Die Darstellung ist nicht vollkommen korrekt im Sinne einer EBNF-Form. Da sie ein Nebenprodukt der Überlegungen bzgl. des Einsatzes von *CUP Parser Generator for Java*^{TM} ist, welches aber beim Netzgraphen doch nicht benutzt wird, ist auch diese Form der Grammatik nicht mehr in Benutzung, so dass sie nicht weiter ausformuliert wurde. Sie dient daher auch nur zur allgemeinen Veranschaulichung des DXF-Formats. Desweiteren bezieht sie sich lediglich auf die für TRACS relevanten Teile.

(daher auch „Vereinfachte“ DXF-Grammatik)

```
<dxf>          ::= [...] /* DXF-Datei-Header, weitere Sektionen */
                   <blocksec>
                   <entitysec>
                   "  0"
                   "EOF"

<blocksec>     ::= "  0"
                   "SECTION"
                   "  2"
                   "BLOCKS"
                   "  0"
                   <blocks>
                   "ENDSEC"

<entitysec>    ::= "  0"
                   "SECTION"
                   "  2"
                   "ENTITIES"
                   "  0"
                   <entities>
                   "ENDSEC"

<blocks>        ::= "BLOCK"
                   "  2"
                   [[:alnum:]_-]+
                   " 70"
                   "0"
                   <startcoord>
                   "  3"
```

```
                    [[:alnum:]_'-']+
                    "  1"
                    " "
                    "  0"
                    {<entities>}
                    "ENDBLK"
                    "  0"

<startcoord>    ::= " 10"
                    -?[[:digit:]]+"."[[:digit:]]+
                    " 20"
                    -?[[:digit:]]+"."[[:digit:]]+
                    " 30"
                    -?[[:digit:]]+"."[[:digit:]]+

<endcoord>      ::= " 11"
                    -?[[:digit:]]+"."[[:digit:]]+
                    " 21"
                    -?[[:digit:]]+"."[[:digit:]]+
                    " 31"
                    -?[[:digit:]]+"."[[:digit:]]+

<entities>        ::= {<line>}
                    {<circle>}
                    {<point>}
                    {<text>}
                    {<insert>}
                    {}

<entheader>     ::= "  8"
                    [[:alnum:]]+
                    " 62"
                    "256"
                    "  6"
                    "ByLayer"
                    <startcoord>

<insert>        ::= <entheader>
                    "  0"

<point>         ::= <entheader>
```

```
                    "  0"

<circle>        ::= <entheader>
                    " 40"
                    [[:digit:]]+.[[:digit:]]+
                    "  0"

<text>          ::= <entheader>
                    " 40"
                    [[:alnum:],.!-_?-]* /* eig.: alle Zeichen erl. */
                    " 50"
                    -?[[:digit:]]+"."[[:digit:]]+
                    " 41"
                    -?[[:digit:]]+"."[[:digit:]]+
                    "  7"
                    [[:alnum:],.!-_?-]* /* eig.: alle Zeichen erl. */
                    " 71"
                    "0"
                    " 72"
                    "1"
                    <endcoord>
                    " 73"
                    "2"
                    "  0"
```

Beim DXF-Format ist zu beachten, dass es zeilenweise aufgebaut ist. Eine zeile enthält einen eindeutigen Code und jede zweite Zeile den dazugehörigen Wert. Es ergeben sich daraus also Zeilenpaare. Mehrere Codes und Werte hintereinander in der selben Zeile sind nicht erlaubt.
Die Anzahl der Zeilen einer DXF-Datei muss immer gerade sein, da die Informationen wie beschrieben immer als Zeilenpaar vorkommen.

Henrik Röhrup

B.3 DXF-Beispiel

Dies ist der Rahmen einer DXF-Datei mit den vier vorkommenden Sektionen **HEADER**, **TABLES**, **BLOCKS** und **ENTITIES**, von denen nur beiden letzten für uns interessant sind:

```
999
dxflib 2.0.2.1
  0
SECTION
  2
HEADER

[..] verschiedene globale Definitionen, etc....
wird von uns nicht weiter beachtet...

  0
ENDSEC
  0
SECTION
  2
TABLES

[..] Definition von Tabellen, u.a. für Layer...
wird von uns nicht weiter beachtet...

  0
ENDSEC
  0
SECTION
  2
BLOCKS

[..] verschiedene Block-Definitionen...

  0
ENDSEC
  0
SECTION
  2
ENTITIES

[..] verschiedene Entities...

  0
ENDSEC
  0
```

```
EOF
```

B.3.1 Beispieldatei

Abbildung B.1 zeigt eine graphische Beispielzeichnung eines Gleisnetzes. Die DXF-Repräsentation der für TRACS interessanten Sektionen von dieser Zeichnung sieht ausschnittsweise wie folgt aus:

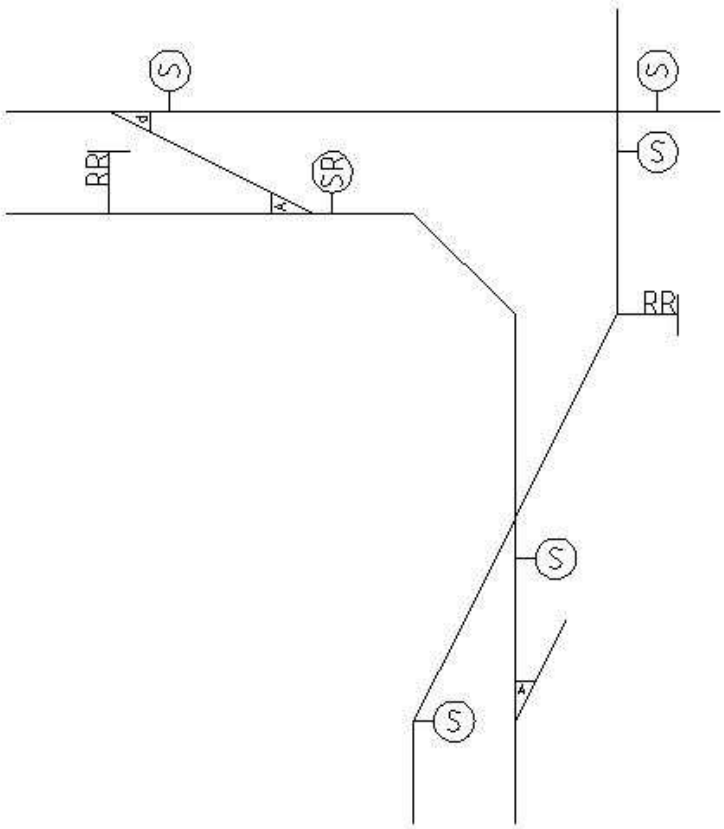


Abbildung B.1: Eine Besipiel-Zeichnung eines Gleisnetzes

B.3.2 Auszug aus BLOCKS-Section

Hier folgt der Auszug aus der zur Beispiel-Zeichnung B.3.1 gehörenden **BLOCKS** Sektion, welche nach Anwendung von *DXFSeperator.java* in die Datei *blocks-tmp.tng* extrahiert wurde:

```
  0
SECTION
  0
BLOCKS
  0
BLOCK
 10
0.0
 20
0.0
 30
0.0
  3
crossing_lr-0
  1

  0
LINE
  8
track
 10
-2.5
 20
0.0
 30
0.0
 11
2.5
 21
10.0
 31
0.0
  0
LINE
  8
track
 10
0.0
 20
0.0
 30
```

```
0.0
 11
0.0
 21
10.0
 31
0.0
  0
ENDBLK

[...] aufgrund der Dateilänge wurde nur ein Ausschnitt gewählt...
weitere Blöcke würden hier folgen...

  0
ENDSEC
```

B.3.3 Auszug aus ENTITIES-Section

Hier folgt der Auszug aus der zur Beispiel-Zeichnung B.3.1 gehörenden Sektion **ENTITIES**, welche nach Anwendung von *DXFSeperator.java* in die Datei *entities-tmp.tng* extrahiert wurde:

```
  0
SECTION
  0
ENTITIES
  0
INSERT
  8
0
  2
crossing_lr-0
 10
80.0
 20
50.0
 30
0.0
  0
INSERT
```

```
  8
0
  2
point_arc_right-0
 10
85.0
 20
60.0
 30
0.0
  0
INSERT
  8
0
  2
crossing_x-0
 10
85.0
 20
70.0
 30
0.0
  0

[..] Aufgrund der Dateilänge wurde hier nur ein Ausschnitt gewählt.
Die eigentliche Datei enthält deutlich mehr Entities (INSERT, LINE,
o.ä.)...

INSERT
  8
0
  2
signal_example-3
 10
80.0
 20
53.0
 30
0.0
  0
ENDSEC
```

Als Anmerkung sei erwähnt, das das Programm **DXFSeperator.java** neben dem Extrahieren der Sektionen auch die vorhandenen Daten etwas ausdünnt und dabei die unwesentlichen Zeilenpaare entfernt.

Henrik Röhrup

Anhang C

DXF-Objekt-Bibliothek

C.1 CAD-Bibliothek

Im Folgenden eine graphische Darstellung aller Elemente der DXF-Objekt-Bibliothek:

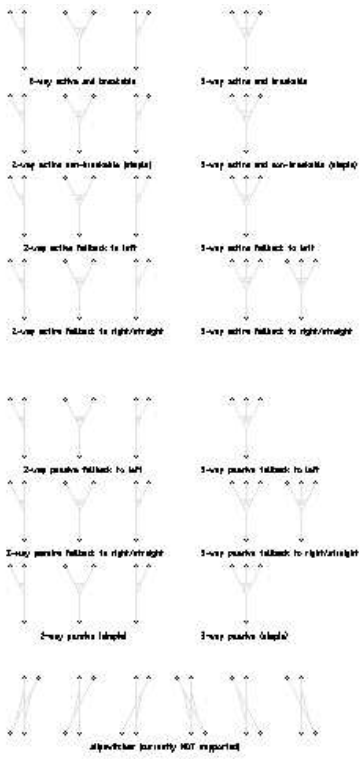


Abbildung C.1: Weichen-Objekte

Die Abbildung C.1 zeigt sämtliche Weichen-Objekte der Teil-Bibliothek **points**. Es gibt diverse 2-Wege-Weichen, 3-Wege-Weichen und sog. Kreuzungsweichen (Kreuzungen mit

Weichenelementen). Die beiden zuerst genannten Kategorien sind sowohl als aktive (mit Motor) wie als passive (Umstellung bei Überfahren) vorhanden.

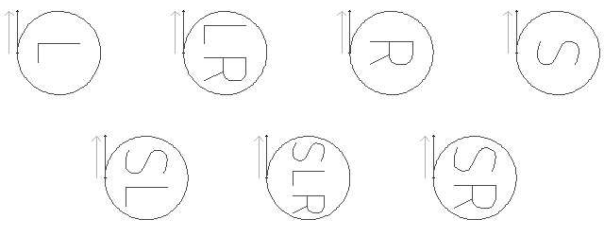


Abbildung C.2: Signal-Objekte

Die Abbildung C.2 zeigt sämtliche Signal-Objekte (Fahrtrichtungssignale) der Teil-Bibliothek **signals**. Welches Signal welche Richtungen anzeigen kann, ist dem Textfeld innerhalb des Kreises zu entnehmen.

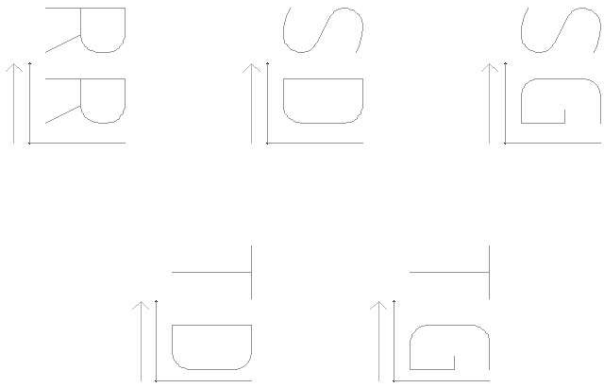


Abbildung C.3: Sensoren-Objekte

Die Abbildung C.3 zeigt sämtliche Sensor-Objekte der Teil-Bibliothek **sensors**. Welches Sensor welche Funktion hat, ist dem Textfeld zu entnehmen. Die Abkürzungen entsprechen den bereits anderweitig im Bericht benutzen Bezeichnungen (z.B. Kapitel 7.5.3.1.4,

Seite 177).

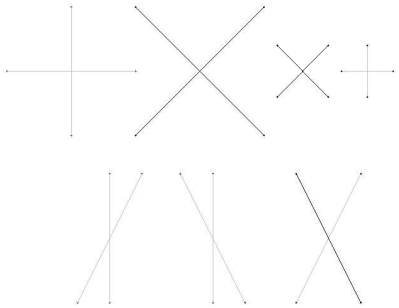


Abbildung C.4: Schienen-Objekte (Kreuzungen)

Die Abbildung C.4 zeigt Kreuzungs-Objekte der Teil-Bibliothek **track**.

Abbildung C.5: Sonstige Objekte

Die Abbildung C.5 zeigt sonstige Schienen-Objekte der Teil-Bibliothek **others**. Die Bedeutung der Elemente ist der Abbildung zu entnehmen.

Anhang D

Netzgraph-Konfigurationsdatei

D.1 Beispiel einer Netzgraph-Konfigurationsdatei

Es folgt ein Ausschnitt aus der Konfigurationsdatei **Separator.conf** für die Netzgraph-Klasse *DXFSeparator.java*:

```
#
# config file for DXFSeparator.class
#
# -lines with leading comment-char (#) are ignored.
# -lines with only blanks dto.
# -empty lines dto.
#
#
# line format as follows:
#
# description of code:code (example: X-Coordinate: 10)
#
# The value before the first ':' is the description of the DXF
# control-code (tag). Every character except ':' is allowed!
# The value between the first and the second ':' is the code/tag
# itself.
# The value after the second ':' is the section-descriptor ('e'
# for ENTITIES and 'b' for BLOCKS). No other characters are allowed!
#
# The control-code consists of 1 to 3 digits. If the number of digits
# is less than 3, the space is filled with the appropriate number of
# blanks BEFORE any digits. No other characters are allowed! Only
```

```
# digits and blanks! Exactly three characters are allowed - no more
# no less!
#

# ENTITIES-SECTION:

control:  0:e
Layername:  8:e
X-coord1: 10:e
X-coord2: 11:e
Y-coord1: 20:e
Y-coord2: 21:e
Z-coord1: 30:e
Z-coord2: 31:e

# BLOCKS_SECTION:

control:  0:b
dummy:  3:b
Textvalue:  1:b
Radius,TextHeight: 40:b
```

Die genaue Beschreibung kann dem Datei-Kopf entnommen werden.

Henrik Röhrup

Anhang E

Netzgraph-Konverter - Element-Listen-Ausgabe

E.1 Beispielgleisnetz

In Abbildung E.1 ist ein komplettes Gleisnetz zu sehen, welches mit Hilfe der CAD-Software QCad eingegeben worden ist:

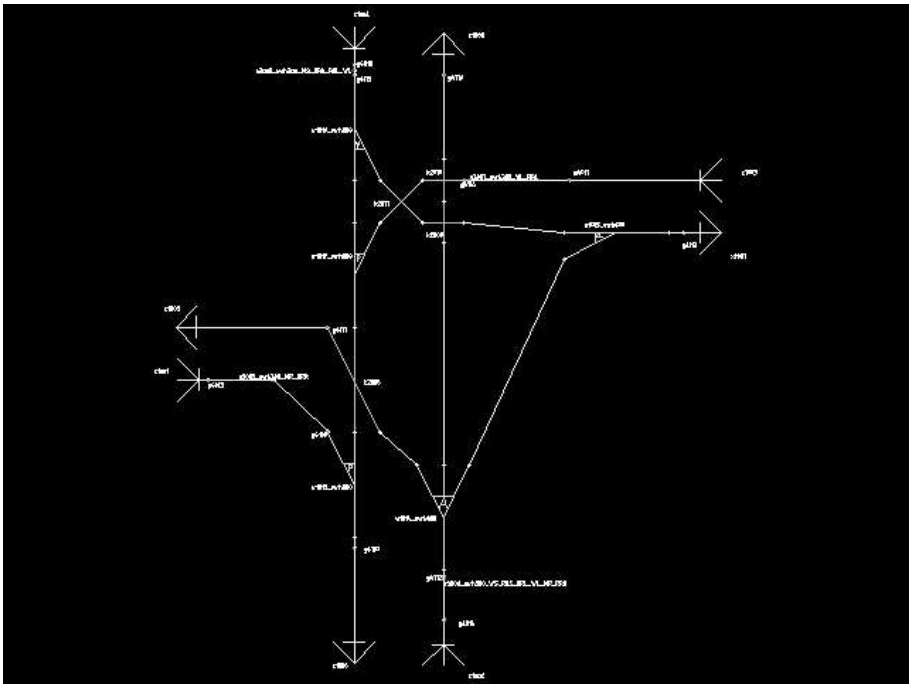


Abbildung E.1: Beispiel-Gleisnetz (Gleisnetz2)

E.2 Element-Listen-Ausgabe

Im Folgenden ist die komplette Ausgabe der Element-Liste eines Gleisnetzes (Abbildung E.1 auf der vorherigen Seite) zu sehen, welche mit dem Parameter „–list“ erzeugbar ist:

```
name       :  s3007
category   :  signals
type       :  signal_l
properties:  WL;RRL
switchtime:  300 ms
coords     :  -19.0:34.0:0.0

name       :  s3008
category   :  signals
type       :  signal_sl
properties:  WS;RRS;RRL;WL
switchtime:  300 ms
coords     :  -30.0:44.5:0.0

name       :  s3005
category   :  signals
type       :  signal_r
properties:  WR;RRR
switchtime:  300 ms
coords     :  -37.8:15.0:0.0

name       :  s3006
category   :  signals
type       :  signal_slr
properties:  WS;RRS;RRL;WL;WR;RRR
switchtime:  300 ms
coords     :  -21.5:-4.0:0.0

name       :  g4114
category   :  sensors
type       :  sensor_tg
coords     :  -19.4:34.0:0.0

name       :  g4116
```

```
category  :  sensors
type      :  sensor_tg
coords    :  -21.5:44.0:0.0

name      :  g4115
category  :  sensors
type      :  sensor_tg
coords    :  -30.0:44.1:0.0

name      :  g4111
category  :  sensors
type      :  sensor_tg
coords    :  -32.5:20.0:0.0

name      :  g4017
category  :  sensors
type      :  sensor_rr
coords    :  -9.4:34.0:0.0

name      :  g4018
category  :  sensors
type      :  sensor_rr
coords    :  -30.0:45.1:0.0

name      :  g4014
category  :  sensors
type      :  sensor_rr
coords    :  -21.5:-7.9:0.0

name      :  g4110
category  :  sensors
type      :  sensor_tg
coords    :  -30.0:-0.9:0.0

name      :  g4109
category  :  sensors
type      :  sensor_tg
coords    :  -32.5:10.1:0.0

name      :  g4013
category  :  sensors
```

```
type       :  sensor_rr
coords     :  -44.0:15.0:0.0

name       :  g4113
category   :  sensors
type       :  sensor_tg
coords     :  1.3:29.0:0.0

name       :  g4112
category   :  sensors
type       :  sensor_tg
coords     :  -21.5:-3.8:0.0

name       :  w1017
category   :  points
type       :  point_sr_passive
properties:  passive
switchtime:  800 ms
coords     :  -30.0:25.0:0.0

name       :  w1016
category   :  points
type       :  point_sl_act
properties:  none
switchtime:  800 ms
coords     :  -30.0:39.0:0.0

name       :  w1013
category   :  points
type       :  point_sl_passive
properties:  passive
switchtime:  800 ms
coords     :  -30.0:5.0:0.0

name       :  w1014
category   :  points
type       :  point_slr_act
properties:  none
switchtime:  800 ms
coords     :  -21.5:1.9000001:0.0
```

```
name       :  w1015
category   :  points
type       :  point_sl_passive
properties:  passive
switchtime:  800 ms
coords     :  -5.0:29.0:0.0

name       :  k2008
category   :  crossings
type       :  crossing_rl
coords     :  -30.0:15.0:0.0

name       :  k2011
category   :  crossings
type       :  crossing_90deg_rotby45deg_small
coords     :  -25.5:32.0:0.0

name       :  k2009
category   :  crossings
type       :  crossing_90deg_small
coords     :  -21.5:30.0:0.0

name       :  k2010
category   :  crossings
type       :  crossing_90deg_small
coords     :  -21.5:34.0:0.0

name       :  x1004
category   :  marks
type       :  entrance
coords     :  -30.0:46.5:0.0

name       :  x1008
category   :  marks
type       :  exit
coords     :  -21.5:46.0:0.0

name       :  x1003
category   :  marks
type       :  entrance
coords     :  3.0:34.0:0.0
```

```
name      :  x1007
category  :  marks
type      :  exit
coords    :  3.0:29.0:0.0

name      :  x1002
category  :  marks
type      :  entrance
coords    :  -21.5:-10.2:0.0

name      :  x1006
category  :  marks
type      :  exit
coords    :  -30.0:-10.0:0.0

name      :  x1001
category  :  marks
type      :  entrance
coords    :  -44.8:15.0:0.0

name      :  x1005
category  :  marks
type      :  exit
coords    :  -45.0:20.0:0.0
```

Anhang F

Glossar

branch - Die Seite einer Weiche, an der mehrere Gleise entspringen, nennt sich Verzweigungsseite (branch) – die andere Seite mit einem Gleis heißt Stammseite (stem).
CAD - s. *Computer Aided Design*
Callback-Funktion - Funktion, dazu bestimmt sind, als Funktionszeiger an einen anderen Programmteil übergeben zu werden und von diesem in klar definierten Situationen aufgerufen zu werden.
Casting - Unter bestimmten Bedingungen kann in einem Programm eine Variable eines gegebenen Typs in einen anderen Typen überführt werden - dies bezeichnet man als Typumwandlung oder Casting.
Central Processing Unit - Central Processing Unit (CPU) heißt übersetzt so viel wie Zentraleinheit. „Im allgemeineren Sinne fasst man unter Zentraleinheit den Rechnerkern und den Hauptspeicher zusammen.“[Lex97, S. 546]
Compiler - Ein Programm, das Text aus einer (Programmier-)Sprache in einen maschinenlesbaren Code übersetzt.
Computer Aided Design - kurz auch CAD genannt - Eine moderne Möglichkeit, grafische Modelle zu entwerfen. Mit Hilfe von Bildbearbeitungsprogrammen, die den Schwerpunkt „technisches Zeichnen“ haben, entstehen Objekte am Computer.
Concurrent Versions System - Concurrent Versions System (CVS) heißt übersetzt so viel wie Versionsverwaltungssystem. Dieses System verwaltet Versionen von Dateien auf einem zentralen Server und erleichtert die Arbeit, wenn mehrere Personen an einer Datei arbeiten und ihre Änderungen zusammenfügen müssen.
CPU - s. *Central Processing Unit*
CUP - CUP Parser Generator for Java™ (aka. Java™ Based Constructor of Useful Parsers) ist ein Tool für Java™, welches aus einer Spezifikationsdatei einen Parser generiert (ähnlich bison für C). Es ist ausgelegt auf eine Kombination mit dem Lexer-Generator *JFlex*.
CVS - s. *Concurrent Versions System*
Data Exchange File - kurz auch DXF - Dateiformat für zweidimensionale *CAD-*

Zeichnungen
Datendarstellung, etikettierte - Beschreibung, in der die einzelnen Elemente durch Auszeichnungen gekennzeichnet sind.
Domain Specific Language - Eine Domain Specific Language (DSL) ist eine spezielle Sprache, die für ein bestimmtes Anwendungsgebiet entwickelt wird. Wir beschreiben mit unserer DSL den Aufbau eines Gleisnetzes, vor allem Sensoren, Signale und Weichen.
Doxygen - Doxygen ist ein Dokumentationswerkzeug für (u. a.) Java™ und C/C++, welches auf Grundlage von Quelltext-Dateien eine Dokumentation (z. B. in HTML) automatisch generiert. Sofern Klassen, Methoden und Variablen im Quellcode kommentiert sind, werden diese auch in die Dokumentation kopiert.
DSL - s. *Domain Specific Language*
DXF - s. *Data Exchange File*
EBNF - s. *Extended Backus-Naur Form*
Extended Backus-Naur Form - Die Extended Backus-Naur Form (EBNF) ist eine gängige Notation, mit der man die Syntax einer Sprache formulieren kann. Wir beschreiben damit die Syntax unserer *DSL*.
EVP - Entwicklungs- Verifikationsprozess.
Fehlerbaumanalyse - Eine Fehlerbaumanalyse (Fault Tree Analysis) ist ein Systemanalyseverfahren, um systematisch darzustellen, wann ein unerwünschtes Ereignis auftreten kann. Dazu werden alle möglichen Ursachen des betreffenden Ereignisses unter Verwendung von Boolescher Algebra miteinander verknüpft.
Garbage Collector - Ein Garbage Collector läuft bei der Ausführung des Java™-Interpreters in regelmäßigen Abständen ab und gibt Speicher, der reserviert ist und nicht mehr referenziert wird, wieder frei.
Hashing - Zugriffseffiziente Verwaltung von Datensätzen.
Hazard - Als Hazard bezeichnet man eine Situation, in der eine Gefahr für Mensch und Material besteht.
HW/SW-Integrationstest - HW/SW-Integrationstests (HSI) testen ein Sub-System, das aus Originalhardware und integrierter Software besteht, in Blackbox-Form.
i386 - Bezeichnung für eine PC-Plattform (kompatibel zu einem Prozessor vom Typ 80386 der Firma Intel)
Java™ - plattformunabhängige Programmiersprache, entwickelt von der Firma Sun Microsystems
Java2™ - Version 2 der Java-Programmiersprache
Java2™-Runtime-Environment - Laufzeitumgebung der Programmiersprache Java™, welche zum Ausführen von Java™-Programmen benötigt wird
Java2™-Software-Developmentkit - Entwicklungsumgebung für Java™, welche zum Übersetzen von Java™-Quellcode benötigt wird
J2RE - s. *Java2™-Runtime-Environment*
J2SDK - s. *Java2™-Software-Developmentkit*
JDK/JRE - s. *Java2™-Software-Developmentkit* und *Java2™-Runtime-Environment*

JFlex - JFlex ist ein Tool für Java™, welches aus einer Spezifikationsdatei einen Lexer generiert (ähnlich flex für C).
Kernel - Systemkern eines Betriebssystems
Konflikt, hart - Ein harter Konflikt bezeichnet die Aufgabelung von zwei *Routen* an einer Weiche.
Konflikt, weich - Ein weicher Konflikt bezeichnet das Zusammentreffen von zwei *Routen* (Zusammenführung an einer Weiche).
Konverter - Ein Konverter ist ein Programm, welches ein Datenformat in ein anderes umwandelt.
Lexer - Ein Lexer (auch Scanner genannt) liest eine Datei Zeichen für Zeichen ein und gibt die erkannten Token an einen Parser weiter.
Linux - verbreitetes freies Unix-ähnliches Betriebssystem für Personal-Computer und Server
Little Endian - Ein Format für die Byte-Reihenfolge für Integer-Werte, bei der das niederwertigste Byte an der niedrigsten Adresse gespeichert wird.
Mac OS - ein Computer-Betriebssystem
Model Checker - Die Aufgabe eines Model Checkers ist zu überprüfen, ob gegebene Eigenschaften in einem gegebenen Zustandsübergangssystem erfüllt sind.
Netzgraph - komplette Beschreibung eines Gleisnetzes (egal, ob physisch vorhanden oder imaginär), enthält Nachbarschaftsbeziehungen, topologische Informationen. Kann u.a. als *CAD*-Zeichnung realisiert sein.
NuSMV - NuSMV ist ein Tool zur Verifikation von Systemen mit Hilfe von Model Checking Techniken. Dokumentation: [CCO+]
Padding - Verlängern von Datenstrukturen mit nicht benutzten Bytes, damit die Adressen im Speicher bestimmten Forderungen von Teilbarkeit durch eine ganze Zahl erfüllen.
Parser - Ein Parser nimmt erkannte Token von einem Lexer entgegen und überprüft die Abfolge der Token darauf, ob sie der Grammatik des Parsers entsprechen.
Point Position Table - Beschreibt die Stellungen von Weichen, die nötig sind, um eine *Route* befahren zu können.
Projektierungsdaten - Eingabedaten für den Steuerinterpreter, die in einem Binärformat vorliegen. Sie werden vom Compiler aus der *DSL* generiert.
Real Time Test Language - Die Real Time Test Language (RTTL) ist eine durch *Verified* entwickelte formale Spezifikationssprache, die für die Beschreibung des Verhaltens von Parallelprozessen unter Echtzeitbedingungen entworfen wurde.
Route - Eine Route ist ein bestimmter Weg durch ein Gleisnetz, definiert über die Abfolge der Sensoren, die der Zug beim Befahren der Route passiert.
Route Conflict Table - Enthält die harten und weichen Konflikte zwischen den *Routen*. Siehe auch *Konflikt, hart* und *Konflikt, weich*.
Route Definition Table - Definiert die Reihenfolge der Sensoren und die Eintrittsbedingungen für jede Strecke.
RT-Tester - Das RT-Tester System (RealTime-Tester) ist zur Durchführung automa-

tischer Hardware-in-the-Loop Tests und Software-Komponententests von eingebetteten Echtzeit-Systemen entworfen worden. Dokumentation: [Ver04]
RTTL - s. *Real Time Test Language*
Scanner - s. *Lexer*
SHM - s. *Shared Memory*
Shared Memory - kurz: SHM - Eine vom Betriebssystem unterstützte Methode zur Kommunikation von mehreren Prozessen untereinander. Hierbei können sämtliche beteiligten Prozesse auf einen gemeinsamen Adressbereich zugreifen.
Signal Setting Table - Beschreibt die Stellung von Signalen, die für die Einfahrt einer *Route* notwendig sind.
Software-Integrationstest - kurz: SWI - Testet Softwarekomponenten, die sich gegenseitig beeinflussen. Dabei wird das Ein- und Ausgabeverhalten der Software getestet.
Spline - Eine Kurve, die durch eine bestimmte Anzahl von Punkten verläuft und diese möglichst „glatt“ miteinander verbindet (Interpolation).
Socket - „Als Socket bezeichnet man eine streambasierte Programmierschnittstelle zur Kommunikation zweier Rechner in einem TCP/IP-Netz. Sockets wurden [...] mit Berkeley UNIX 4.1/4.2 allgemein eingeführt. Das Übertragen von Daten über eine Socket-Verbindung ähnelt dem Zugriff auf eine Datei.“[Krü00, S. 1057]
stem - s. *branch*
Steuerinterpreter - Ein Programm, das die *Projektierungsdaten* interpretiert und die Steuerungsaufgaben für verschiedene Hardware-Elementen erfüllt.
SWI - s. *Software-Integrationstest*
Systemintegrationstest - Systemintegrationstests testen ein komplettes System innerhalb seiner operationalen Umgebung oder einer Simulation davon.
Test - Überprüfung, ob das implementierte Steuerungssystem die erwartete Aufgabe erfüllt.
TND - s. *Tram Network Description*
Toggle-Sensor - Sensor, der bei jeder Aktivierung ein Bit an der Schnittstelle zum *Steuerinterpreter* abwechselnd auf 0 oder 1 setzt.
TRACS - s.*Train Control Systems*
Train Control Systems - Train Control Systems (TRACS) ist der Name eines Projektes an der Universität Bremen, das im Zeitraum vom Wintersemester 2003/2004 bis zum Sommersemester 2005 von zehn (ursprünglich ca. 20) Studenten im Hauptstudium durchgeführt wurde. „Dieses Projekt beschäftigt sich mit der Entwicklung und Verifikation von Bahnsteuerungssystemen, wie sie in Stellwerken und Zügen zum Einsatz kommen.“[TRA]
Tram Network Description - kurz: TND - vom *TRACS*-Projekt entworfene *DSL*, die zur Beschreibung von Gleisnetzen dient.
Übersetzer - s. *Compiler*
Unix - ein Computer-Betriebssystem
Validierung - Nachweis, dass geforderte Eigenschaften eines Systems gegeben sind,

wird durch Untersuchungen erbracht.
Verified - Der Hersteller des *RT-Testers*. [Ver]
Verifikation - Nachweis, dass jedes einzelne Entwicklungsprodukt mit der Spezifikation konsistent ist.
Verschlusstabellen - Diese Tabellen beschreiben die Bedingungen, die notwendig sind, um die sichere Durchfahrt aller *Routen* zu gewährleisten. Es gibt folgende Tabellen: *Point Position Table*, *Route Conflict Table*, *Route Definition Table*, *Signal Setting Table*.
Windows™ - kommerzielles Betriebssystem für Personal-Computer der Firma Microsoft®

Literaturverzeichnis

[AD94] Alur, Rajeev und David L. Dill: *A Theory of Timed Automata*. In: *Theoretical Computer Science, Volume 126, No 2*, 1994.

[ASU99] Aho, A. V., R. Sethi und J. D. Ullmann: *Compilerbau Teil 1*. Oldenbourg, München, Zweite Auflage, 1999. ISBN: 3-486-25294-1.

[Aut03] Autodesk, Inc., San Rafael, Kalifornien, USA: *DXF Reference 2004*, 2003. http://adeskftp.autodesk.com/prodsupp/downloads/dxf.pdf.

[Bre04] *Eine elektrisierende Ausstellung*, 2004. http:// www.bsag.de/6128.php, Abruf am 07.05.2004.

[CCO+] Cavada, R., A. Cimatti, E. Olivetti, G. Keighren, M. Pistore und M. Roveri: *NuSMV 2.2 User Manual*. http://nusmv.irst.itc.it.

[Cho78] Chow, Tsun S.: *Testing Software Design Modeled by Finite-State Machines*. In: *IEEE Transactions on Software Engineering, SE-4(3), pp. 178-186*, März 1978.

[Cod] *IO-Warrior Universeller IO-Controller am USB*. http://www.codemercs.com/IOWarriorD.html. Heruntergeladen am 7. Januar 2005.

[Dec] *PCI 8255/8254 48 x I/0 + Timer Karte*. http://www.decision-computer.de/dio/dio8255pci.htm. Heruntergeladen am 7. Januar 2005.

[Fre04] *Freunde der Bremer Straßenbahn e.V.*, 2004. http://www.fdbs.net, Abruf am 07.05.2004.

[HP02] Haxthausen, A. E. und J. Peleska: *A domain specific language for railway control systems*. In: *Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology – IDPT02*, Pasadena, Kalifornien, USA, Juni 2002.

[HP03a] Haxthausen, A. E. und J. Peleska: *Automatic Verification, Validation and Test for Railway Control Systems based on Domain-Specific Descriptions – IFAC03*. In: Tsugawa, S. und M. Aoki (Herausgeber): *Proceedings of the 10th IFAC Symposium on Control in Transportation Systems*, Tokio, Japan, 2003. Elsevier Science Ltd, Oxford.

[HP03b] Haxthausen, A. E. und J. Peleska: *Generation of Executable Railway Control Components from Domain-Specific Descriptions*. In: Tarnai, G. und E. Schnieder (Herausgeber): *Formal Methods for Railway Operation and Control Systems: Proceedings of Symposium – FORMS03*, Seiten 83–90, Budapest, Ungarn, Mai 2003.

[Ins92] Institut zur Entwicklung moderner Unterrichtsmedien e. V., Bremen: *Mikroprozessor- und Mikrocomputertechnik*, Erste Auflage, 1992.

[Krü00] Krüger, G.: *GoTo Java 2 – Handbuch der Java-Programmierung*. Addison Wesley, München, Zweite Auflage, 2000. ISBN: 3-8273-1710-X.

[Krü03] Krüger, G.: *Handbuch der Java-Programmierung*. Addison Wesley, München, Dritte Auflage, 2003. ISBN: 3-8273-2120-4.

[Lex97] Lexikonredaktion, Meyers (Herausgeber): *Schüler Duden Informatik*. Meyers Lexikonredaktion, Mannheim, Dritte Auflage, 1997. ISBN: 3-411-04483-7.

[MDB03] Mühlbacher, D., P. Dobrovka und J. Brauer: *Computerspiele - Design und Programmierung*. mitp, Bonn, Zweite Auflage, 2003. ISBN: 3-8266-0920-4.

[Mus04] Mustun, A.: *QCad Benutzerhandbuch*. RibbonSoft, Winterthur, Schweiz, 2004. http://www.ribbonsoft.com/qcad/manual_reference/de/.

[Pel02] Peleska, J.: *Formal Methods for Test Automation – Hard Real-Time Testing of Controllers for the Airbus Aircraft Family*. In: *Proceedings of the Sixth Biennial World Cobference on Integrated Design and Process Technology – IDPT02*, Pasadena, Kalifornien, USA, 2002.

[Pel03] Peleska, J.: *Automated Testsuites for Modern Aircraft Controllers*. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, Seiten 1–10. Shaker, Aachen, 2003.

[PGHD04] Peleska, J., D. Grosse, A. E. Haxthausen und R. Drechsler: *Automated Verification for Train Control Systems*. In: Schnieder, E. und G. Tarnai (Herausgeber): *Formal Methods for Automation and Safety in*

Railway and Automotive Systems, Braunschweig, Germany, 2004. Technical University of Braunschweig.

[PT02] Peleska, J. und A. Tsiolakis: *Automated Integration Testing for Avionics Systems*. In: *International Conference on Software Testing – ICSTEST*, Düsseldorf, 2002.

[PZ] Peleska, J. und C. Zahlten: *Safety Critical Systems*. http://www.informatik.uni-bremen.de/agbs/lehre/ws0405/scs1/hintergrund/safety.ps.

[PZ99] Peleska, J. und C. Zahlten: *Test Automation for Avionic Systems and Space Technology (Extended Abstract)*. Presented at the Workshop of the GI Working Group Test, Analysis and Verification of Software, 1999.

[Sun] Sun Microsystems Inc.: *Java™ 2 Platform, Standard Edition, v 1.4.2 API Specification*. http://java.sun.com/j2se/1.4.2/docs/api/ Heruntergeladen am 3. Oktober 2004.

[TRA] *TRACS - Train Control Systems*. http://www.informatik.uni-bremen.de/agbs/lehre/tracs.

[Ver] *Verified Systems International GmbH*. http://www.verified.de.

[Ver04] Verified Systems International GmbH, Bremen: *RT-Tester 6.0 User Manual Issue 1.0*, 2004. http://www.verified.de.