AUTOMATED VERIFICATION FOR TRAIN CONTROL SYSTEMS

Jan Peleska[1], Daniel Große[1], Anne E. Haxthausen[2] and Rolf Drechsler[1]
[1] *Universität Bremen, TZI*
Address: TZI, P.O. Box 330440, D-28334 Bremen, Germany
Phone: +49-421-218-7092, Fax: +49-421-218-3054 ,
E-Mail: {drechsle,grosse,jp}@tzi.de
[2] *Technical University of Denmark, Informatics and Mathematical Modelling*
Address: IMM/DTU, building 322, DK–2800 Kgs.Lyngby, Denmark
Phone: +45-45-257510, Fax: +45-45-930074 , E-Mail: ah@imm.dtu.dk

Abstract: In this paper we present an approach for automated verification of train or tram control systems which is based on model checking. The strategies and techniques applied are distinguished from comparable concepts of other authors in the following ways: (1) The control model to be verified is equivalent to executable machine code. As a consequence, there is no need for stepwise refinement and associated formal verification, in order to establish the conformance between the model and the executable software. (2) The well-known state explosion problems occurring frequently when checking complete railway control models are overcome by a combination of bounded model checking and inductive or compositional reasoning. (3) Since the verification approach is fully automated, it can be applied to each concrete control system, instantiated from a generic system using configuration data encoding the railway network and the routes to be controlled. This offers an alternative to more conventional (semi-formal) verification strategies, where the generic system is fully verified (so-called type certification), but only partial verification is exercised on the concrete instances.
Keywords: railway control systems, domain-specific language, formal methods, bounded model checking

1. INTRODUCTION

Motivation. The development of safety-critical railway control systems is usually based on generic software code, applicable to a well-defined variety of control tasks and configurations. For a concrete system – e.g., the interlocking controller for the railway network of a specific train station – the generic code is instantiated with configuration data describing the railway network topology and the user-specific operational requirements, such as pre-defined train routes through the network and additional site-specific safety constraints.

The conventional verification approach follows this generic development paradigm: A major verification effort is focused on the so-called *type certification*, where it is shown that the generic code is really suitable for the intended class of instantiations. For concrete systems, a more restricted verification suite concentrates on checking the correctness of configuration data and on the verification of new software developed for customised user requirements. At least for German railways, verifications are performed in a semi-formal way, using a combination of reviews, inspections and testing. Formal verification is only applied in a very limited way.

Now practical experience shows that this verification approach should be improved, because the concrete configured systems quite frequently exhibit faulty behaviour. Analysis of these malfunctions indicates that there are three main causes for these occurrences of errors: (1) The generic system is not formally verified, that is, no theorem has been established showing that the generic code will really operate correctly for all admissible instances. As a consequence, new errors are often discovered in the generic code when new variants of configuration data are used for the first time. (2) Even a full proof covering all possible configurations cannot anticipate the different hardware configurations concrete systems will operate in. As a consequence, a second class of errors is caused by the violation of hardware constraints, such as CPU power, memory capacity or hardware interface latency. (3) The restrictions to be observed by legal instances of the generic code are not specified in a formal way, so that a mechanised exhaustive verification of configuration data cannot be performed. Therefore a third class of errors is caused by incompatibilities between configuration data, generic code and available hardware, as well as inconsistencies within the configuration data itself.

Due to these considerations, the authors advocate an alternative verification approach: (A) For the generic system, the verification investigates mainly whether the generic code is sufficiently general to meet all functional instantiation requirements. This demonstrates the usability without attempting to establish any safety-related properties. (B) For each concrete system instance, the reusable code and the associated configuration data are formally verified, thereby establishing the correct implementation of both safety and user requirements. (C) For demonstrating the compatibility of hardware and embedded software, hardware-in-the-loop testing is performed.

While the advantages of our approach with respect to the correctness of concrete systems are obvious, a critical issue remains to be solved: If the effort for performing the verification steps (B) and (C) is too high, it will be infeasible to perform the large number of concrete system verifications. Therefore it is a crucial aspect of our strategy that the steps (B) and (C) should be performed in an automated way.

Conceptual Model for Railway Control Systems. A well-accepted conceptual model for the domain of railway control systems is depicted in *Figure 1*. The *domain of control* – also called *physical model* – consists of the railway network portion under consideration and the trains moving within. The railway network is composed of track segments and track elements, the latter are points, signals and sensors indicating trains passing at known locations. Trains enter the domain of control "non-deterministically", that is, governed by rules exercised by neighbouring domains. These rules ensure that no safety requirements can be violated when a train enters the domain of control, but otherwise our domain has no influence on the points in time and the frequency of their occurrence.

While inside the domain's network portion, the *controller* shall protect trains against hazardous situations. To this end, the controller implements behaviour specified in the *control model*: The controller observes sensor state changes, deriving from them the current train locations within the network. Trains may issue requests to pass through the network on pre-define routes. The controller issues commands to switch signal and point states and monitors the correct state of these track elements.

Hazardous situations are characterised by states in the physical model where a *safety condition* Φ is violated. Typically, Φ involves train locations, the directions they are moving in and point positions. Formally speaking, the domain of control D and controller C represent two communicating concurrent sub-systems. The design objective for C is to restrict the possible state transitions of D, when operating in the parallel configuration $(D \| C)$, so that each reachable state of D within $(D \| C)$ fulfills Φ. The concrete safety requirements encoded in Φ depend on the applicable rules within the domain of control. Tram control systems have less stringent rules than railways, and additional rules may be specified for concrete network instances, such as train stations.

Observe that trains only have to be protected from entering hazardous states while residing within the domain of control. After having passed along a finite number of track segments, each train will either leave the network portion under consideration or stop in a stable safe state.

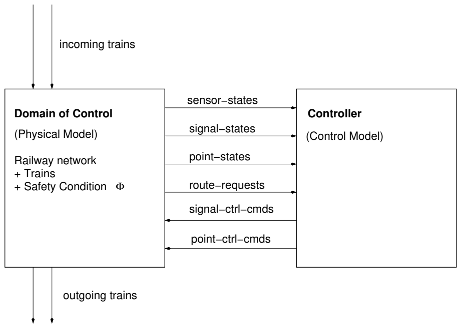

Fig. 1. Control model for the railway domain.

Development and Verification Approach. The underlying approach for system development and verification comprises the following steps:

(1) As inputs to the development process, we require the specification of the railway network portion under consideration (i.e., track segments and elements) and a set of tables describing the train routes through the network (see *Section* 2).

(2) A generator applying generic rules for the behaviour of trains moving through networks automatically creates the concrete transition relation for the operational semantics of trains moving within the domain of control given by the concrete railway network. Moreover, this generator creates the safety predicate Φ and the proof obligations to be verified in order to establish validity of Φ in all situations (*Section* 5).

(3) A second generator instantiates the concrete transition rules for the control model from a generic controller description. To this end, we exploit a previous result (Haxthausen and Peleska, 2003*b*) where it has been shown that the instantiated code and configuration data – if developed according to a domain-specific framework for railway control systems – can be directly interpreted as a timed state-transition system. As a consequence, it is unnecessary to abstract the executable code and data to a higher-level representation, in order to obtain a model which is fit for model checking: The code and data may be checked directly, or after application of a one-to-one adaption to the specific syntactic requirements of the model checker. As a consequence, no justification is required with respect to abstract interpretations of code and their suitability to uncover certain classes of errors, such as safety violations.

(4) As a main verification objective, the steps (1) to (3) result in the obligation to show that the transition rules for the control model are sufficient to prove the safety condition Φ for the parallel composition $(D \| C)$ of concrete physical and control model. This is performed by bounded model checking.

(5) The transition rules of the control model are mapped to executable code. This process is supported by a specification of the available hardware resources which contains, for example, a mapping from abstract interface variables to concrete hardware interface drivers.

(6) Automated hardware-in-the-loop tests are exercised on the integrated HW/SW system.

Related Work. The approach to derive executable, semantically well-defined code for railway control systems from domain-specific descriptions has been published by the authors in (Peleska *et al.*, 2000; Haxthausen and Peleska, 2003*b*; Haxthausen and Peleska, 2000*b*; Haxthausen and Peleska, 2002). A prototype code generator following this approach has been implemented (Haxthausen *et al.*, 2004). The complete verification strategy comprising steps (A) to (C) has been introduced in (Haxthausen and Peleska, 2003*a*). Alternative verifications methods for the railway domain which are based on theorem proving have been considered in (Haxthausen and Peleska, 2000*a*). The inductive reasoning style applied there is reused in the present paper to establish a verification strategy which is suitable for the application of bounded model checking (BMC) techniques. The general BMC approach is described in (Biere *et al.*, 1999; Biere *et al.*, 2003).

The application of model checking to problems of railway control has been investigated by several authors; we name (Damm *et al.*, 1999) as an example. It is a well-known problem that conventional model checking approaches in this domain frequently fail due to state explosions, when applied to railway control systems of realistic size. This has been observed, for example, by (Clarke *et al.*, 2001), where the authors also suggest BMC techniques to overcome these difficulties.

Overview. To illustrate the application of model checking techniques, we investigate a case study about a tram control system which is introduced in *Section 2*. The focus on trams is more suitable for a short presentation than a railway control system, since the set of safety conditions is smaller: For trams it is not necessary to consider shunting and flank protection. The domain of control D, its operational execution rules and the associated safety requirements Φ applicable for the case study are formally described in *Section 3*, using description techniques for timed state transition systems (TSTS) introduced in (Haxthausen and Peleska, 2000*b*; Haxthausen and Peleska, 2002). The associated controller model C – i. e., the representation of the executable control system software – is shown in *Section 4*. While it is our strategy to generate this code automatically from domain-specific descriptions, the development technique applied is not relevant for the verification techniques introduced in this paper: We only rely on the availability of TSTS representations which are known to be equivalent to the executable machine code. In *Section 5* the bounded model checking approach is illustrated, using the case study modelled in the previous sections. *Section 6* contains the conclusion.

2. CASE STUDY – TRAM CONTROL SYSTEM

In this section, we introduce the domain of control and the associated requirements for the tram controller in an informal way. Formal representations follow in subsequent sections.

Figure 2 shows the static part of the domain of control. The track elements – signals, points and sensors – are marked by Sxy, $Wabc$ and $Gxy.z$, respectively. Track segments within the domain of control are identified by sensor pairs which also assign a fixed direction of driving trams to the segment: The pair $(G20.2, G21.0)$ denotes the middle segment on the left-hand track, to be passed by trams in North-South direction, starting at $G20.2$ and ending at $G21.0$. The segments where trams enter the domain of control are identified by their destination sensor. For example, the North-South track on the left-hand side may be entered from a neighbouring domain at $(-, G20.1)$. Analogously, segments leaving the domain of control are denoted by their source sensor, as in $(G21.1, -)$. Signals are positioned at sensors and apply to the driving direction of the associated segment. Each segment is associated with a maximal number of trams which are allowed to reside simultaneously on the same segment, when driving in the same direction; for the case study, we allow at most one tram per segment.

Observe that the places where two track segments meet in the direction of driving trams are depicted light gray: These places do not represent switchable points, but solid track components where trams may enter a new track segment from another one.

The network should have the property that any potential route through the network going from one of the designated entry sensors to an exit boarder sensor respects the driving directions of the involved segments. Furthermore, the set of entry sensors and exit sensors are disjoint.

The user requirements define a collection of *routes* along which trams should cross the network. These routes are defined in a *route definition table*. An example is shown in *Table 1*.

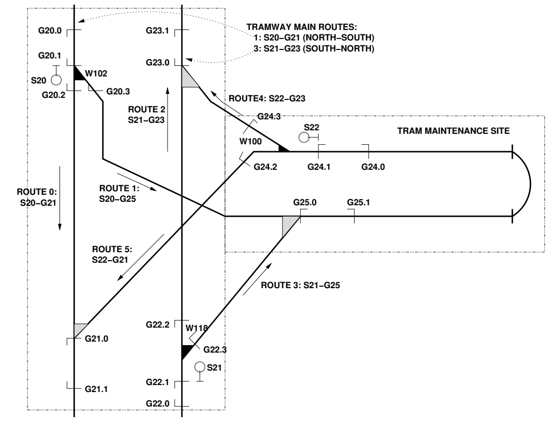

Fig. 2. Network description – the static part of the physical model.

Route definition table	
Route	**Route Sensor Sequence**
0	$\langle G20.1, G20.2, G21.0, G21.1\rangle$
1	$\langle G20.1, G20.3, G25.0, G25.1\rangle$
2	$\langle G22.1, G22.2, G23.0, G23.1\rangle$
3	$\langle G22.1, G22.3, G25.0, G25.1\rangle$
4	$\langle G24.1, G24.3, G23.0, G23.1\rangle$
5	$\langle G24.1, G24.2, G21.0, G21.1\rangle$

Table 1. Route definition table.

Each route is represented as a sequence of neighbouring sensors. To enter a route, certain conditions on signal and point states must be satisfied, and other routes currently occupied by trams must not cross the new route and introduce collision hazards. Only if these requirements are fulfilled, the entering signal of a route switches to "GO" and the waiting tram may enter the requested route. The requirements for each route are specified as part of the configuration data and may be represented by *Tables 2-4*.

Point position table			
Route	**W100**	**W102**	**W118**
0	—	straight	—
1	—	left	—
2	—	—	straight
3	—	—	right
4	right	—	—
5	straight	—	—

Table 2. Point position table.

Signal setting table		
Route	**Signal**	**Setting**
0	S20	go-straight
1	S20	go-left
2	S21	go-straight
3	S21	go-right
4	S22	go-right
5	S22	go-straight

Table 3. Signal setting table.

Route conflict table						
Route	**Conflicts with**					
	0	1	2	3	4	5
0		●				○
1	●		○	○		○
2		○		●	○	○
3		○	●			
4			○			●
5	○	○	○		●	

Table 4. Route conflict table.

The static part of the physical model is complemented by the behavioural model, described by a cartesian product space containing state vectors[1] and rules for transition between them. The components of state vectors describe the actual states of signals, points and sensors, as well as the number of trams residing on each track segment. For transitions between state vectors, the following rules – described more formally below – have to be observed: (1) A tram may be placed on a segment entering the domain of control whenever this place is empty. (2) When passing along a point from stem to branch, trams will enter the segment connected to the branch associated with the current point position. (3) A tram located in front of a signal in HALT state will not move. (4) A tram located in front of signal in GO state may or may not move. (5) A tram located at a signal at an entry point to one or more routes may request to pass along any of these routes. (6) A tram does not change driving direction.

The safety requirements Φ can be specified informally by the following rules:

(SF1) No two trams are simultaneously driving in opposite directions on the same segment. (SF2) No two trams are simultaneously driving towards the same sensor from different directions. (This holds for sensors connecting two segments as well as for sensors like G21.0 connecting 3 segments.) (SF3) For each segment the number of trams residing simultaneously on it is less than a predefined maximum *max* (which is chosen to be 1 in our case study). (SF4) For each point the number of trams that drive simultaneously on its branches is less *max*. (SF5) There are not trams residing simultaneously on two different segments that cross each other.

Note that due to the network assumptions (SF1) is trivially fulfilled.

3. MODELLING THE DOMAIN OF CONTROL

In this section we present the behavioural part of the physical model as a timed state transition system (TSTS) denoted by D, using a notation introduced in (Haxthausen and Peleska, 2003*b*). The TSTS semantics is *not* based on the maximal progress principle: Applicable transitions may be applied, but instead the time tick might be incremented. If several rules apply to the same component, a rule may be selected in a non-deterministic way. However, transitions marked as `urgent` *must* be taken whenever they may fire in the current state.

In the context of the case study presented here we focus on normal (not failing) behaviour of the track elements.

3.1 Notation

In the following let $PointState$ = { LEFT, STRAIGHT, RIGHT }, $SigState$ = { HALT, GO-LEFT, GO-STRAIGHT, GO-RIGHT } and $SenState$ = { HIGH, LOW } be the respective element state values.

All track elements and routes are identified by numbers $id \in \mathbb{N}$. For the concrete system to be analysed in our case study, the identifier sets are instantiated by the concrete ranges of numbers required for signals, points, sensors and routes:

$SigId = \{0(= S20), 1(= S21), 2(= S22)\}$

$PointId = \{0(= W100), 1(= W102), 2(= W118)\}$

$SenId = \{0(= G20.0), \ldots, 17(= G25.1)\}$

$RouteId = \{0, \ldots, 5\}$.

3.2 State space SD

The global state SD of the domain of control is a cartesian product

$$\begin{aligned} SD \quad = \quad & TM \times \\ & SIG \times SIGREQ \times SIGREQTM \times \\ & PT \times PTREQ \times PTREQTM \times \\ & SEN \times SENTM \times SENCNT \end{aligned}$$

with typical vector

$$(t, actsig, reqsig, reqsigtm, \\ actpt, reqpt, reqpttm, sen, sentm, c)$$

and product components

$$\begin{aligned} TM &= \mathbb{N} \\ SIG &= SigId \rightarrow SigState \end{aligned}$$

[1]Observe that the term *configuration* is often used in the literature in place of *state vector*, when explaining the operational semantics of specification formalisms. We prefer the latter term within this paper, in order to avoid confusion with the different possible *configurations* of the tram control systems under consideration.

$$
\begin{aligned}
SIGREQ &= SigId \rightarrow SigState \\
SIGREQTM &= SigId \rightarrow \mathbb{N} \\
PT &= PointId \rightarrow PointState \\
PTREQ &= PointId \rightarrow PointState \\
PTREQTM &= PointId \rightarrow \mathbb{N} \\
SEN &= SenId \rightarrow SenState \\
SENTM &= SenId \rightarrow \mathbb{N} \\
SENCNT &= SenId \rightarrow \mathbb{N}
\end{aligned}
$$

with the following interpretation: $t : \mathbb{N}$ denotes the current clock tick. $actsig(s) : SigState$ denotes the actual state of signal s. $reqsig(s) : SigState$ denotes the requested state for the signal. The point in time when this request has been issued by the controller is stored in $reqsigtm(s)$. For each point, p, its actual state $actpt(p) : PointState$, its requested state $reqpt(p) : PointState$ and the request time $reqpttm(p)$ are defined analogously. For each sensor, g, $sen(g) : SenState$ denotes its actual state, $sentm(g) : \mathbb{N}$ a time stamp defined when the transition into HIGH state occurred. $c(g) : \mathbb{N}$ is a "virtual" (not physically existing) counter, the role of which is to indicate the number of trams that have passed the sensor. From the counters one can for instance calculate the number of trams driving on a segment from one sensor towards another sensor using discrete flow equations similar to continuous flow equations known from fluid mechanics. The counters are useful for modelling conditions for when the sensors can go from LOW to HIGH state and for modelling the safety requirements.

3.3 Parallel Composition of Components

The TSTS for the domain of control is the parallel composition of the TSTS describing the behaviour of the time ticks and each signal, point and sensor. There are no additional components modelling trams, since their movements through the tramway network are reflected by the sensor counter state changes (vector component $c : SenId \rightarrow \mathbb{N}$). As described in (Haxthausen and Peleska, 2003*b*), the transition rules for the composite system are obtained by simultaneous application of the transition rules applicable for each signal, point and sensor in the current state. We will see below, that all assignments to variables during a transition are atomic, so that this simultaneous application of transition rules is well-defined by the statement that each interleaving of simultaneous assignments is possible.

3.4 Transition relations

The transition relation of D is defined as the concurrent composition of transition relations for each sensor, point and signal, and a time ticks described below.

Transition relation for time ticks. We model the passage of time as a discrete increment of the TM state component.

```
if (true) { t = t + 1; }
```

Transition relation for signals. The following transition rule is instantiated for each $s \in SigId$. It states that if the actual state is different from the requested state, it will reach the requested state when `delta_s` time units have elapsed after the request was issued.

```
if ( t == reqsigtm(s) + delta_s and
     reqsig(s) != actsig(s) ) {
  urgent : actsig(s) = reqsig(s);
}
```

Expressions like `actsig(s) = reqsig(s)` denote function overriding: The state change caused by the transition results in a new function $actsig : SigId \rightarrow SigState$ which differs from the previous at argument s, where it maps to value $reqsig(s)$.

Transition relation for points. The transition rule for a point is similar to that of a signal, except that the relevant point state component is changed instead of a signal state component and another deadline constant `delta_p` is used.

Transition relation for sensors. For each sensor $g \in SenId$ there are two transition rules, one for going from state LOW to state HIGH and one for going from state HIGH to state LOW.

The "HIGH-to-LOW" rule has the following form:

```
if ( t == sentm(g) + delta_l and
     sen(g) == HIGH ) {
  urgent : sen(g) = LOW;
}
```

It states that when the sensor has been stable in the HIGH state for `delta_l` time units (so that the controller has a chance to detect the HIGH state) it becomes LOW.

The form of the "LOW-to-HIGH" rule depends on the sensor location: (1) network entry, (2) network exit, (3) between two neighbouring segments without a point and (4) at stem and branches of a controllable point, or (5) at stem and branches of a non-controllable point. Furthermore, it depends on whether a signal is associated with the sensor or not.

Due to the usual space limitations, we only give the form for the "LOW-to-HIGH" transition rules of a few of the above mentioned cases.

For an entry border sensor g having a (single) neighbouring sensor $g2$ the form is:

```
if ( t > sentm(g) + delta_tram and
     sen(g) == LOW and
     c(g) == c(g2)) {
  sen(g) = HIGH;
  c(g) +=1;
  sentm(g) = t;
}
```

The rule states that the sensor g may perform a transition to state HIGH, incrementing its counter, and updating its time stamp, when it has been in state LOW for at least `delta_tram` time units (so that the physical requirement *"there is a minimum amount of time which has to pass before the next tram reaches a sensor"* is fulfilled) and its counter has the same value as the counter of its neighbour (so that the physical requirement *"a tram only enters an empty entry segment* $(g, g2)$*"* is fulfilled).

This rule applies for instance to $g = 1 = G20.0$ and $g2 = 2 = G20.1$ for our case study.

For all rules the pattern of actions is the same, however the conditions differ. For a route entry sensor g like $G20.1$ which is in the front of a point, the form of the rule is:

```
if ( sen(g) == LOW and
     c(g) < c(g1) and
     actsig(s) != HALT and
     c(g) == c(g2) + c(g3) ) {
  sen(g) = HIGH;
  c(g) += 1;
  sentm(g) = t;
}
```

where s is the signal associated with the sensor, $g1$ is the preceeding sensor, and $g2$ and $g3$ are the following sensors (e.g. $s = S20.0$, $g1 = G20.0$, $g2 = G20.2$ and $g3 = G20.3$ for $g = G20.1$). Here the conditions express that the counter of the sensor is less than the counter of the preceeding sensor (meaning that a tram is approaching g from $g1$) and the signal is not showing HALT. Moreover, the position at g must not be blocked by another tram, which is expressed by the condition $c(g) = c(g2) + c(g3)$, meaning *"All trams having passed g have also passed either $g2$ or $g3$."*.

4. CONTROL MODEL

In this section we present the controller model, C, in the form of a timed state transition system. The controller consists of a parallel composition of

- a *route dispatcher*, RD, that registers route requests and makes reservations of requested routes in compliance with the requirement that two conflicting routes must never be reserved at the same time,
- for each route, r, a *route controller*, $RC(r)$, that is responsible for setting points and signals during allocation, use and de-allocation of the route, and,
- for each sensor, g, a *counter controller*, $CC(g)$, that each time a tram passes the sensor increments a control counter associated with the sensor.

In a real-world application there would be a safety monitor as well capturing exceptional behaviour.

4.1 State space SC

The controller shares all state components of SD, except sensor counters $SENCNT$ with the domain of control. Technically, this means that the controller acts on a *device abstraction layer* (see (Haxthausen and Peleska, 2002)) encapsulating the interface drivers and providing an interface abstraction equivalent to TM, SIG, ... to the control software. To this end, hardware interface boards provide dual-ported RAM interfaces to the control software, where the state of each controlled device is visible as a static *actual state data structure*. Requests from control software to interface drivers are placed into *request data structures* within the dual ported RAM. As soon as the requested state differs from the current state of a signal or point, the associated drivers issue the corresponding switching commands to the device.

For example, signal states are visible to the control software in read-only mode as an array

```
SigState actsig[NUM_SIGS];
```

with `NUM_SIGS = #SigId`. This array is updated by the device abstraction layer. It can be identified with the state component $actsig : SigId \rightarrow SigState$ in a direct way. State requests are written by the control software to an array

```
SigState reqsig[NUM_SIGS];
```

which is identified with $reqsig : SigId \rightarrow SigState$.

The controller also has a number of additional state components (to be explained below) which are not shared with the physical model. Hence, the state space of the controller is:

$$\begin{aligned} SC \quad = \quad & \ldots \times \\ & CCTR \times RTREQ \times RTRES \times \\ & RCMV \times CCMV \end{aligned}$$

where ... represents the shared components. The new product components are

$$\begin{aligned} CCTR &= SenId \rightarrow \mathbb{N} \\ RTREQ &= RouteId \rightarrow Bool \\ RTRES &= RouteId \rightarrow Bool \\ RCMV &= RouteId \rightarrow RCMode \\ CCMV &= SenId \rightarrow CCMode \end{aligned}$$

where

$$\begin{aligned} RCMode &= \{FREE, ALLOCATING, \\ & \quad ALLOCATED, OCCUPIED\} \\ CCMV &= \{READY, COUNTED\} \end{aligned}$$

are the sets of possible control modes for route controllers and counter controllers, respectively. A typical state vector is:

$$(..., cc, req, res, rc\_cmv, cc\_cmv)$$

The interpretation of the new components is as follows: For any sensor g, $cc(g)$[2] is the internal image of the counter $c(g)$ introduced in the domain of control. $req(r) : Bool$ and $res(r) : Bool$ indicate whether route r has been requested and reserved, respectively. $rc\_cmv(r)$ holds the current control mode of route controller $RC(r)$ for route r. $cc\_cmv(g)$ holds the current control mode of counter controller $CC(g)$ associated with sensor g.

4.2 Transition relations

Observe that all transitions of the controller are urgent, so we drop this key word in the following transition rules.

Route dispatcher. The route dispatcher has a transition rule for each route. It applies these in a specific order, this may be expressed by using a C-style for-loop:

```
const int NUM_ROUTES 6;

for (r=0; r < NUM_ROUTES; r++) {
  if ( req(r) and ~res(r) and
       forall r1 in {1..NUM_ROUTES} .
         ~(conflict(r,r1) and res(r1))
     ) {
    res(r) = TRUE; req(r) = FALSE;
  }
}
```

Observe that - though written in a C-style notation - the specification is not code, but a set of transition rules associated with an application order. The constants reflect instantiations of the transition rules, as

[2] The sensor counters used in the model for the domain of control are "virtual" state components which have been introduced to model the "flow" of trams through the network. Counter values cannot be read directly from the sensor equipment but have to be derived by the control software from the trace of the HIGH/LOW state changes of physical sensors. As a consequence, the controller manages the additional state component $CCTR$.

derived from the domain-specific description of the concrete system. Likewise the *conflict* relation is derived from domain-specific description (the route conflict table).

The rule for a route r states that when the route is requested and not yet reserved and none of its conflicting routes are reserved then the route may be reserved and the request deleted.

Route controllers. Any route controller $RC(r)$ for a route r goes through four possible control modes and has five transition rules.

Initially it is in mode FREE. The first rule states that it may perform a transition from the FREE mode to the ALLOCATING mode by initiating allocation of the route, if the route is reserved and all its conflicting routes marked with $\circ$ in the route conflict table have their entry signal on HALT. Initiating the allocation means to request the points of the route to be switched as specified in the point position table.

The second rule states that it may perform a transition from the ALLOCATING mode to the ALLLOCATING mode by requesting the entry signal of the route to show the GO setting described in the signal setting table if the points of the route have reached their requested settings.

The third rule states that it may perform a transition from the ALLOCATING mode to the ALLLOCATED mode if the entry signal of the route has reached its requested setting and the entry sensor is HIGH (i.e. the tram touches the sensor).

The fourth rule states that it may perform a transition from the ALLLOCATED mode to the OCCUPIED mode by requesting the entry signal to show HALT, if the entry counter is equal to the sum of the counters of the immediate neighbour sensors (i.e. the nose of tram has passed the second sensor of the route).

The fifth rule states that it may perform a transition from the OCCUPIED mode to the FREE mode by deallocating the route (i.e. decrementing the counters of the route with the amount of trams that have left the route and removing the reservation), if the second and the last counters of the route have the same values and the entry signal has reached its requested state HALT. The former condition ensures that there are no trams between the second sensor and the last sensor of the route, and the latter condition ensures that there are no trams between the two first sensors. Together this means that the route is free.

As an example the transition rules for the route controller of route 0 is as follows:

```
if ( rc_cmv(0) == FREE and
     res(0) and
     actsig(S22) == HALT) {
  reqpt(W102) = STRAIGHT;
  reqpttm(W102) = t;
  rc_cmv(0) = ALLOCATING;
}

if ( rc_cmv(0) == ALLOCATING and
     actpt(W102) == STRAIGHT ) {
  reqsig(S20) = GO-STRAIGHT;
  reqsigtm(S20) = t;
}

if ( rc_cmv(0) == ALLOCATING and
     actsig(S20) == GO-STRAIGHT and
     sen(G20.1) == HIGH ) {
  rc_cmv(0) = ALLOCATED;
}

if ( rc_cmv(0) == ALLOCATED and
     cc(G20.1) ==
       cc(G20.2) + cc(G20.3) ) {
  reqsig(S20) = HALT;
  reqsigtm(S20) = t;
  rc_cmv(0) = OCCUPIED;
}

if ( rc_cmv(0) == OCCUPIED and
     cc(G21.1) == cc(G20.2) and
     actsig(S20) == HALT ) {
  res(0) = FALSE;
  rc_cmv(0) = FREE;
  cc(G20.0) = cc(G20.0) - cc(G20.2);
  cc(G20.1) = cc(G20.1) - cc(G20.2);
  cc(G20.2) = 0;
  cc(G21.0) = 0;
  cc(G21.1) = 0;
}
```

Counter controllers. Any counter controller $CC(g)$ for a sensor g goes cyclically through its two possible control modes. It has the following two transition rules:

```
if ( cc_cmv(g) == READY and
     sen(g) == HIGH ) {
  cc(g) += 1; cc_cmv(g) = COUNTED;
}

if ( cc_cmv(g) == COUNTED) and
     sen(g) == LOW ) {
  cc_cmv(g) = READY;
```

```
}
```

The first rule states that the counter controller may perform a transition from the READY control mode into the COUNTED mode by incrementing the control counter by one, but only if the sensor to which it belongs is HIGH. When the sensor becomes LOW again, the counter controller may perform a transition back to the READY mode.

5. VERIFICATION BY BOUNDED MODEL CHECKING

In this section we describe the formal verification approach to system verification. We briefly explain the basic ideas of bounded model checking and then introduce the modelling language SystemC and the formalism for specifying properties. Next our SAT based property checker for SystemC designs is described. The property checker is an extenstion of the work in (Große and Drechsler, 2003) and supports a larger set of SystemC constructs due to a new frontend (Fey *et al.*, 2004). Then the verification approach is demonstrated for parts of the case study modelled in previous sections.

5.1 Bounded Model Checking

In *Model Checking* (also called *Property Checking*) for a given system properties are formulated in a dedicated "verification language". It is then formally proven whether these properties hold under all circumstances. While "classical" CTL-based model checking (Burch *et al.*, 1990) can only be applied to medium sized designs, approaches based on *Bounded Model Checking* (BMC) as discussed in (Biere *et al.*, 1999) give very good results when used for complete blocks with up to 100k gates. In BMC the properties only argue over a finite interval. BMC has originally been proposed for circuit verification and in this context considering a finite number of steps is reasonable. The underlying techniques are outlined below.

5.2 Specification Languages

In the following systems are modelled in SystemC (Grötker *et al.*, 2002). Therefore, first a short overview on SystemC is given. Then the formalism for specification of temporal properties is described.

5.2.1 SystemC

The main features of SystemC for modelling a system are based on the following:

- Modules are the basic building blocks for partitioning a design. A module can contain processes, ports, channels and other modules. Thus, a hierarchical design description becomes possible.

- Communication is realized with the concept of interfaces, ports and channels. An interface defines a set of methods to access channels. Through ports a module can send or receive data and access channel interfaces. A channel serves as a container for communication functionality, e.g. to hide communication protocols from modules.

- Processes are used to describe the functionality of the system, and allow expressing concurrency in the system. They are declared as special functions of modules and can be sensitive to events, e.g. an event on an input signal.

- Hardware specific objects are supplied, like e.g. signals, which represent physical wires, clocks, and a set of data-types useful for hardware modelling.

Besides this, SystemC provides a simulation kernel. The functionality is similar to traditional event-based simulators. Note that a SystemC description can be compiled with a standard C++ compiler to produce an executable specification. The output of a system can be textual, using C++ routines like `cout` for instance, or waveforms. As a C++ class library SystemC can easily be extended by using the facilities of C++.

5.2.2 Property Language

Describing temporal properties for verification can be done in many different ways, since there exist several languages and temporal logics. We use the notation of the property checker from Infineon Technologies AG (see e.g. (Johannsen and Drechsler, 2001; Bormann and Spalinger, 2001) for more details). A property consists of two parts: a list of assumptions (*assume part*) and a list of commitments (*proof part*). An assumption/commitment has the form

```
   at t+a:  expression;
or during[t+a,t+b]:  expression;
or within[t+a,t+b]:  expression;
```

where `t` is a time point, and `a`, `b` $\in \mathbb{N}$ are offsets. If all assumptions hold, all commitments in the proof part have to hold as well. Since `a` and `b` are finite a property argues only over a finite interval, which is called *observation window*.

Example 1 *The property* `test` *says that whenever signal* x *becomes* 1*, two clock cycles later signal* y *has to be* 2.

```
theorem test is
assume:
  at t: x = 1;
prove:
  at t+2: y = 2;
end theorem;
```

In general a property states that whenever some signals have a given value, some other (or the same) signals assume specified values. Of course it is also possible to describe symbolic relations of signals. Furthermore the property language allows to argue over time intervals, e.g. that a signal has to hold in a specified interval. This is expressed by using the keywords `during` and `within`, whereas `during` states that the expression has to hold all the times in the interval and with `within` the expression has to hold at least once in the specified interval. Also a set of advanced operators and constructs is provided to allow for expressing complex constraints more easily.

5.3 Property Checker

The initial sequential property checking problem is converted into a combinational one by unrolling the design, i.e. the current state variables are identified with the previous next state variables of the underlying finite state machine (FSM). The process of unrolling is shown in *Figure* 3.

A BMC instance b of a property P arguing over the finite interval $[t, t+c]$ for a design D is given by:

$$b = \bigwedge_{j=0}^{c-1} T_\delta(\, i(t+j), s(t+j), s(t+j+1)\,) \\ \wedge \neg P(\, i(t), s(t), o(t), \ldots, \\ i(t+c), s(t+c), o(t+c)\,)$$

with

- $i(t) = (i_1^t, \ldots, i_m^t)$ inputs at time point t,
- $s(t) = (s_1^t, \ldots, s_n^t)$ states at time point t,
- $o(t) = \lambda(i(t), s(t))$ outputs at time point t and
- T_δ the transition relation.

The BMC instance b depends only on the states $s(t)$ and the inputs $i(t), \ldots, i(t+c)$. It is unsatisfiable if for all states $s(t)$ and all input sequences $i(t), \ldots, i(t+c)$ the property P over the interval $[t, t+c]$ holds for the design D. If b is satisfiable a counter-example for the property P has been found.

The SystemC property checker takes the FSM representation of the SystemC design and a property as input. Then the property is translated into an expression using only inputs, states and outputs of the SystemC design annotated with time points. The unrolled FSM representation and the property expression are converted into a bit-level representation. Here hashing and merging techniques for minimisation are used. The bit-level representation is given to the SAT solver zchaff (Moskewicz *et al.*, 2001) which has been integrated into the property checker. In case of a counter-example a waveform in VCD format is generated to allow for an easy debugging.

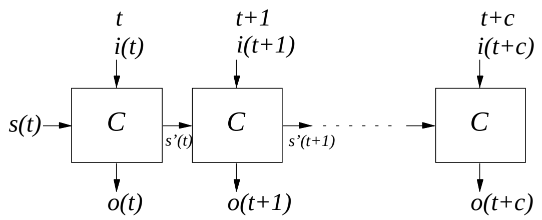

Fig. 3. Unrolling.

5.4 Verification

Examples for the SystemC representation of transition rules as used within the case study have been shown already in *Sections* 3 and 4[3]. In the subsequent paragraphs, we describe the verification steps performed on these models.

Since all of our crucial safety properties are invariants P, they can be verified according to the following inductive principle: (1) Show that P holds after system initialisation. (2) Assume that P holds at t. (3) Prove that P holds at $t+1$.

We illustrate the verification strategy by means of the portion of the safety property Φ dealing with rule (SF5) – *"there are not trams residing simultaneously on two different segments that cross each other"* – introduced in *Section* 2. For our concrete domain of control, a crossing exists between segments $(G20.3, G25.0)$ and $(G22.2, G23.0)$. A tram residing on a segment is modelled by corresponding counter differences: $c(G20.3) > c(G25.0)$ models *"tram resides on* $(G20.3, G25.0)$ and $c(G22.2) > c(G23.0)$ models *"tram resides on* $(G22.2, G23.0)$. Therefore, Φ requires at all times t

$$\neg\,(c(G20.3) > c(G25.0) \wedge c(G22.2) > c(G23.0))$$

In order to prove this, we first relate the "virtual" counters $c(g)$ of the physical model to the internal counters $cc(g)$ of the controller by proving the following property via BMC for each sensor $g \in \{G20.0, G20.1, \ldots\}$:

```
theorem th_counter is
assume:
during[t,t+1]:
   <...additional properties...>
at t+1:
   (c(g) = cc(g))
    or ( sen(g) = HIGH
         and prev(sen(g)) = LOW
         and c(g) = cc(g) + 1 );
prove:
during [t+2,t+4]:
   (c(g) = cc(g))
    or ( sen(g) = HIGH
         and prev(sen(g)) = LOW
         and c(g) = cc(g) + 1 );
end theorem;
```

This theorem establishes an invariant (since the "additional properties" not shown in an explicit way have been established as invariants before) stating that the virtual sensor counters $c(g)$ of the domain of control either carry the same values as the counters $cc(g)$ managed by the controller, or a $LOW \rightarrow HIGH$ transition of g has just occurred, so that $c(g) = cc(g) + 1$. If this is the case, values will coincide again by the next time tick. To establish this property for all sensors, our model checker requires 12s on a personal computer with 1.6GHz CPU.

With this theorem at hand, we can relate the counter relations referenced in Φ to the internal route controller states OCCUPIED and ALLOCATED. Similar to the theorem above, we prove by BMC that

$$(c(G20.3) = c(G25.0)) \vee (c(G20.3) > c(G25.0) \wedge rc\_cmv(1) \in \{ALL., OCC.\})$$

and

[3] In the examples given, we allowed for some minor syntactic deviations from machine-readable SystemC, in order to improve readability.

$$(c(G22.2) = c(G23.0)) \vee \\ (c(G22.2) > c(G23.0) \wedge \\ rc\_cmv(2) \in \{ALL., OCC.\})$$

are invariants. The associated BMC runs take 3s per route. This leaves us with a final theorem to be established, showing that routes 1 and 2 can never be allocated or occupied at the same time. Again, the verification takes about 3s.

6. CONCLUSION

In this paper, an automated verification strategy for railway or tram control systems based on bounded model checking has been described. The invariant nature of safety requirements is well-suited for inductive reasoning over the transition system structure, establishing safety at time t and inferring that the safety invariant still holds at $t+1$. As a consequence, it is not required to explore complete histories starting from system initialisation, as typically performed by classical model checking. Instead, the bounded model checking approach is applied to the inductive verification over a limited number of time steps. This approach allows for mechanised verification of much larger models without running into state explosion problems.

Future work. In sections 3 and 4 it has been explained how the transition rules for the domain of control (D) and the controller (C) can be systematically derived from a domain description (a network description and four route tables). In a similar way proof obligations can be systematically derived. Future work includes implementing generators performing each of these derivation tasks. Since, an XML format has already been defined for domain descriptions, cf. (Haxthausen *et al.*, 2004), a possibility would be to implement the generators as transformations using the Extensible Stylesheet Language XSL that is associated with XML.

REFERENCES

Biere, A., A. Cimatti, E. Clarke, O. Strichman and Y. Zhu (2003). *Bounded Model Checking*. Vol. 58 of *Advances in Computers*. Academic press.

Biere, A., A. Cimatti, E.M. Clarke, M. Fujita and Y. Zhu (1999). Symbolic model checking using SAT procedures instead of BDDs. In: *Design Automation Conf.*. pp. 317–320.

Bormann, J. and C. Spalinger (2001). Formale Verifikation für Nicht-Formalisten (Formal verification for non-formalists). *Informationstechnik und Technische Informatik* **43**, 22–28.

Burch, J.R., E.M. Clarke, K.L. McMillan and D.L. Dill (1990). Sequential circuit verification using symbolic model checking. In: *Design Automation Conf.*. pp. 46–51.

Clarke, E., A. Biere, R. Raimi and Y. Zhu (2001). Bounded model checking using satisfiability solving. *Formal Methods in System Design* **19**, 22–28.

Damm, W., G. Döhmen and J. Klose (1999). Secure decentralized control of railway crossings. In: *Fourth International ERCIM Workshop on Formal Methods in Industrial Critical Systems*. pp. 115–132.

Fey, G., D. Große, T. Cassens, C. Genz, T. Warode and R. Drechsler (2004). ParSyC: an efficient SystemC parser. In: *Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*.

Große, D. and R. Drechsler (2003). Formal verification of LTL formulas for SystemC designs. In: *IEEE International Symposium on Circuits and Systems*. pp. V:245–V:248.

Grötker, T., S. Liao, G. Martin and S. Swan (2002). *System Design with SystemC*. Kluwer Academic Publishers.

Haxthausen, A. E. and J. Peleska (2000*a*). Formal Development and Verification of a Distributed Railway Control System. *IEEE Transaction on Software Engineering* **26**(8), 687–701.

Haxthausen, A. E. and J. Peleska (2000*b*). Formal Methods for the Specification and Verification of Distributed Railway Control Systems: From Algebraic Specifications to Distributed Hybrid Real-Time Systems. In: *Forms '99 - Formale Techniken für die Eisenbahnsicherung Fortschritt-Berichte VDI, Reihe 12, Nr. 436*. VDI-Verlag, Düsseldorf. pp. 263–271.

Haxthausen, A. E. and J. Peleska (2002). A Domain Specific Language for Railway Control Systems. In: *Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology, (IDPT2002), Pasadena, California*.

Haxthausen, A. E. and J. Peleska (2003*a*). Automatic Verification, Validation and Test for Railway Control Systems based on Domain-Specific Descriptions. In: *Proceedings of the 10th IFAC Symposium on Control in Transportation Systems*. Elsevier Science Ltd, Oxford. ISBN 0-08-044059-2.

Haxthausen, A. E. and J. Peleska (2003*b*). Generation of Executable Railway Control Components from Domain-Specific Descriptions. In: *Proceedings of the Symposium on Formal Methods for Railway Operation and Control Systems (FORMS'2003)*. L'Harmattan Hongrie. pp. 83–90.

Haxthausen, A. E., N. Christensen and R. Dyhrberg (2004). From Domain Model to Domain-specific Language for Railway Control Systems. In: *Proceedings of Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2004)), Braunschweig, Germany*.

Johannsen, P. and R. Drechsler (2001). Formal verification on register transfer level – utilizing high-level information for hardware verification. In: *IFIP Int'l Conf. on VLSI*. pp. 127–132.

Moskewicz, M.W., C.F. Madigan, Y. Zhao, L. Zhang and S. Malik (2001). Chaff: Engineering an efficient SAT solver. In: *Design Automation Conf.*. pp. 530–535.

Peleska, J., A. Baer and A. E. Haxthausen (2000). Towards Domain-Specific Formal Specification Languages for Railway Control Systems. In: *Proceedings of the 9th IFAC Symposium on Control in Transportation Systems 2000, June 13-15, 2000, Braunschweig, Germany*. pp. 147–152.