AUTOMATIC VERIFICATION, VALIDATION AND TEST FOR RAILWAY CONTROL SYSTEMS BASED ON DOMAIN-SPECIFIC DESCRIPTIONS

Anne E. Haxthausen * and **Jan Peleska** **

* *CSE/IMM, Technical University of Denmark, DTU, Bldg. 321, DK-2800 Lyngby, e-mail: ah@imm.dtu.dk*
** *TZI, Universität Bremen, P.O. Box 330440, D-28334 Bremen, e-mail: jp@tzi.de*

Abstract: In this paper we present a method for the verification of route based railway control systems developed from domain-specific descriptions following an approach suggested by the authors in earlier publications. A domain-specific description consists of a network description, a route description and a hardware description. The network description is given a state transition system semantics that describes the dynamic behaviour of the uncontrolled domain and in terms of which one can specify the safety requirements. From the route description and hardware description one can automatically derive an executable system also having a state transition system semantics. The safety properties of the executable software can be automatically verified by model checking, and the executable integrated hardware/software system can be automatically tested.

Keywords: formal methods, verification, test, safety, railways, tramways.

1. INTRODUCTION

The task of railway and tramway control systems is to ensure safe operation of the railway/tramway traffic so that catastrophes like train/tram collisions and derailments are avoided. Hence, the correctness of such control systems is highly critical. The European CENELEC standards recommend the use of formal methods for verification and test to ensure safety when developing railway applications of highest safety level.

In (Haxthausen and Peleska, 2003) we suggested a formal development method based on the following ideas: for each control system to be developed, application-specific parameters are specified in a domain-specific language, and from this specification a control system is automatically generated and verified to be safe. We have developed the domain-specific language so that it adopts terms and concepts from the application domain, thereby facilitating the communication between application experts and the specialists developing computerised control systems.

In this paper we focus on our approach for verification, validation and test of the generated control systems. In other papers (Haxthausen and Peleska, 2002; Haxthausen and Peleska, 2003), the domain-specific language is presented and it is described how control systems are automatically generated from domain-specific specifications. The present paper focuses on tramways. The same concepts can also be applied for railways, but the concrete safety requirements as well as the capabilities of sensors, points and signals differ.

In Section 2, we outline our development method, and then, in Section 3 we elaborate on the technical details of the modelling, verification and test phases. Finally, in Section 4, we state our conclusions.

2. DEVELOPMENT, VERIFICATION, VALIDATION AND TEST CONCEPT

In this section we sketch the interaction between processes for development, verification, validation and test (VVT). A graphical illustration is given in Figure 1.

The development of an application consists of three major steps:

Creation and checking of a domain-specific description. In the first major step a domain-specific specification is created. It consists of a *network description* (a graph representing the topology of the track elements, their positions and interconnections), a *hardware description* (specifying the hardware types of the track elements and the target controller hardware) and a *route description* (tables describing tram routes through the network). Examples of such descriptions can be found in (Haxthausen and Peleska, 2003). It is checked by a tool that the specifications are statically well-formed.

The network description is given a state transition system (STS) semantics that describes the dynamic behaviour of the uncontrolled domain. The considered events include tram movements and uncontrolled switching of points and signals. As in (Haxthausen and Peleska, 2000), it can be specified what it means for the system to be in a *safe* state.

The route description is implicitly given a state transition system (STS) semantics by a transformation into an execution description as explained in next paragraph.

Generation and verification of an executable system – software integration. An automatic transformation compiles the route description and the hardware description into configuration data for an *execution description* that consists of (assembler/C) programming language patterns. From the perspective of object orientation, the code patterns are instantiated by the configuration data, thereby resulting in an executable software system. The executable code consists of re-usable *interpreters* acting on the configuration data.

The semantics of the execution description is that of a state transition system (STS) describing the dynamic behaviour of the controlled domain, i.e. the possible switching sequences for signals and points which allow trams to traverse the network along the intended routes. The configuration data specifies the "dimension" of the state space (e. g., the number of routes through the network), the *transition conditions* and the *effect* of the transitions performed on the state space. A special state encoding for trams, sensors, signals and points ensures that only a very limited set of integer operations is required to implement the evaluation of transition conditions and the effect of state transitions. The control structures are restricted to loops and if-else statements. Because of the simplicity of the STS semantics and the restrictive use of machine operations we expect that a full proof can be established showing that all possible executions of the instantiated code patterns are consistent with the STS specified by the configuration data – this task is currently performed in our research groups.

As a consequence, it remains to be shown that the configuration data only admits executions which are safe with respect to the network description (STS), when executed by the given interpreters. This task can be performed via model checking in combination with exhaustive testing. Since verification can be performed exhaustively, it is not required to verify the transformation from route description to execution description: Any transformation error would be revealed in the checking process.

HW/SW integration and testing. The last development step consists in loading the re-usable STS interpreter software together with driver software and configuration data into the controller hardware. To this end, the hardware description is used. Even in the presence of fully verified safety control software and configuration data there remains a risk for potential failure of the integrated HW/SW system: We do not expect that it will be feasible to verify all driver software, firmware and processor micro instructions formally and complete. Moreover, the allocation of software and data on hardware may be faulty. As a consequence, HW/SW integration tests must be performed. The test cases can be derived automatically in analogy to the model checking process. Moreover, real-time test execution and evaluation is possible using test automation systems.

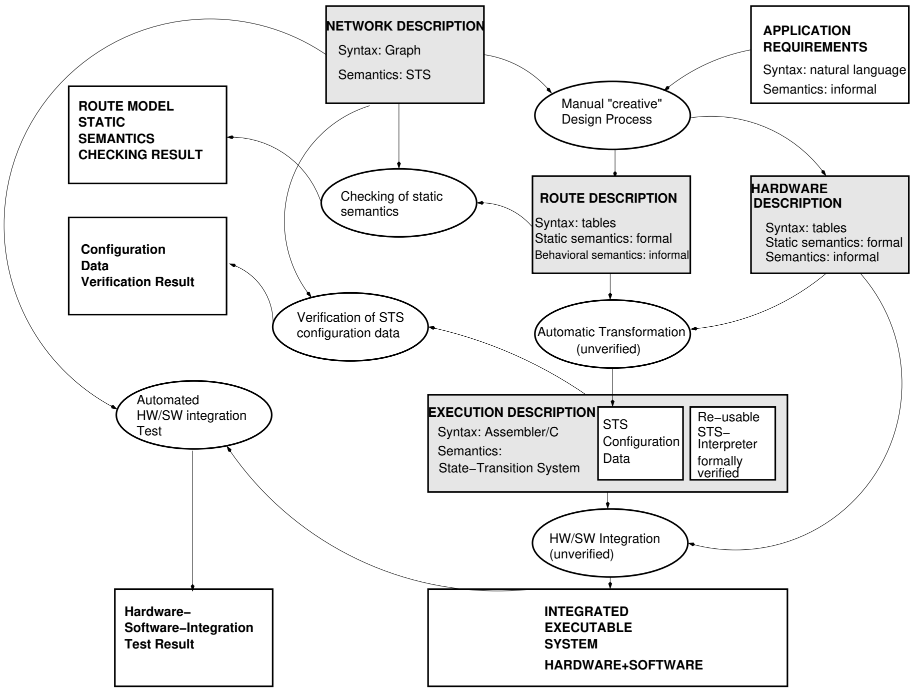

Fig. 1. Development,verification, validation and test processes.

3. TECHNICAL DETAILS

The purpose of this section is to give technical details of the verification and test phases, as well as the models involved. In (Haxthausen and Peleska, 2003) we have introduced a state transition system notation, in order to capture the behavioural semantics of the control system in a formal way. This notation is also convenient for introducing a behavioural semantics for the network description, describing how trams move along the railway network and which states of the network are considered to be safe. For a detailed introduction of the notation, readers should consult the above publication.

3.1 Overall Verification Approach

We follow the established modelling approach for safety-critical systems, by defining a global state space S, a physical model (the network description NTS) and a controller model CTS. This results in a state transition system $(NTS \| CTS)$ acting on S. A subset of S is shared by NTS and CTS: The former writes variables representing physical observables (states of sensors, signals and points) while the latter reacts on changes in these observables by setting control variables for switching signals and points. The safety conditions are invariants of S. The verification problem is that of a *reachability analysis*: Prove that for any legal initial state, an execution of $(NTS \| CTS)$ will never reach a state violating the safety conditions.

We first verify (this will be described in a future paper), that, for any instance of configuration data, the interpreter code implements the state transition system semantics correctly. Therefore configuration data can be verified by model checking on a formal representation of $(NTS \| CTS)$.

If the correctness of the configuration data has been established, we perform a hardware-software integration test, in order to establish that the interpreter software will still implement the STS semantics correctly when embedded into the specific hardware and interface driver configuration. If model checking for the configuration data is performed using forward reachability techniques, the associated data transitions may be used directly for the production of test data sequences and expected results.

3.2 STS Semantics for Network Descriptions

The network description specifying the track topology and the associated track elements (points, signals, sensors) gives rise to a *behavioural physical model* (which we still call the network description) specifying how trams may move along this network. Observe that the original static network description already defines the behavioural model in a unique way by means of the construction rules given in the paragraphs below.

3.2.1. Static relations. The static network description defines the following functions, which are considered as constants referenced in the STS definition given below: Let G be the set of sensor identifiers, Sid the set of signal identifiers and Pid the set of point identifiers as given by the network description. The Boolean function $\eta \in G \times G \rightarrow Bool$ gives the neighbour relationship for sensors, $signal \in G \times G \tilde{\rightarrow} Sid$ is a partial function for which $signal(g1, g2)$ gives the identifier of the signal (if any) at $g1$ in direction $g2$, $stem\_sensor \in Pid \rightarrow G$ and $branch\_sensors \in Pid \rightarrow \mathbb{P}(G)$ are functions giving the sensors on the stem side and branching sides of a given point, $connects \in (Pid \rightarrow PointState) \rightarrow (G \times G \rightarrow Bool)$ is a function for which $connects(\omega)(g1, g2)$ is true if there is a connection from $g1$ to $g2$ when the points are in the point states given by ω. Crossing track segments are captured by a boolean function $cross : (G \times G) \times (G \times G) \tilde{\rightarrow} Bool$. In addition, we mark the sensors where trams enter/leave the network under consideration by $enter : G \rightarrow Bool$ and $exit : G \rightarrow Bool$. We assume that the enter/exit track elements of the network are never stems or branches of points, and that $enter(g) \Rightarrow \neg exit(g)$ and $exit(g) \Rightarrow \neg enter(g)$.

We use the following abbreviations for element states:

H, GL, GS, GR denote signal states HALT, GO-LEFT, GO-STRAIGHT, GO-RIGHT,
HI,LO denote sensor states HIGH or LOW,
PL, PR, PS denote point positions LEFT, RIGHT or STRAIGHT
U, F denote track element (point, signal, sensor) states UNDEFINED or FAILURE.
UHI, ULO denote sensor states UNDEFINED-TOWARDS-HIGH or UNDEFINED-TOWARDS-LOW.

We use sets $PointState = \{PL, PS, PR, U, F\}$, $SigState = \{H, GL, GS, GR, U, F\}$ and $SensState = \{HI, LO, UHI, ULO, F\}$ for the respective element states.

3.2.2. Parallel structure of the network description. The *network transition system (NTS)* is a state transition system over the state space S specified below. NTS is a parallel composition

$$NTS = (G_1 \| \dots \| G_g \| PT_1 \| \dots \| PT_p \| SIG_1 \| \dots \| SIG_s)$$

where SIG_i are state transition systems over S modelling the behaviour of each signal i, PT_j model the behaviour of each point j and G_k model the behaviour of each sensor k. The transition relation T of NTS is determined by the transition relations of the individual STS, using the $|$-construction defined in (Haxthausen and Peleska, 2003).

3.2.3. State space S. We define $S = SENS \times P \times SIG \times TM \times CTRL$. Intuitively speaking, $SENS$ captures the sensor states and records in which direction the neighbouring track segments are currently used by trams, P reflects the point states and SIG the signal states. Set $TM = \mathbb{N}$ models discrete time ticks. The $CTRL$-state components represent the internal state variables of the controller CTS which are not shared with the physical model NTS. The details related to NTS are elaborated in the subsequent paragraphs.

3.2.3.1. Sensor states SENS. Define $SENS = G \rightarrow (SensState \times CTR \times DST \times GTM)$ with $CTR = \mathbb{N}$, $DST = G \cup \{\epsilon\}$ and $GTM = \mathbb{N}$: An element $\alpha \in SENS$ is a function mapping each sensor g to a 4-tuple $\alpha(g) = (x, n, d, u)$. Component x represents the physical sensor state and value u denotes the point in time when the transition into state x occurred. The intuitive interpretation, which will be formally introduced in the transition rules below is as follows: Whenever a tram passes sensor $g \in G$, the sensor will perform a state transition $LO \longrightarrow HI$, and fall back to LO again after having been stable for at least a minimum interval. The transition from LO to HI may be performed via an additional transient state UHI where the proper sensor value cannot be determined (e. g. because a debouncing operation has to be performed). Similarly, the transition from HI to LO may be performed via an additional transient state ULO. A correctly operating sensor will stay in UHI or ULO for only a limited time amount δ_u, otherwise it turns into state failure. Moreover, the HI- and LO-states will be stable long enough for the control system to detect this state (minimum time duration δ_c). Therefore the associated time stamps u help to model the transition relations with respect to timing assumptions. The physical state component x is augmented by auxiliary components n, d which are useful for modelling safety conditions: Component n is a counter which is incremented on each transition into the HI state. If component d is the identifier of a neighbouring sensor $d \in G$, the intuitive interpretation is *"tram drives from g to d"*. Therefore d is called *(current) destination sensor* of g. Value $d = \epsilon$, indicates that no tram drives from g to any of its neighbours. Given $\alpha \in SENS$, we use abbreviations $ctr : G \rightarrow CTR$

and $dst : G \to DST$ for the projection of α on its counter and destination component, respectively.

3.2.3.2. Point states P. Define $P = Pid \to (ReqPointState \times PointState \times PTM)$, where $PTM = \mathbb{N}$ and $ReqPointState = \{PS, PL, PR\}$. An element $\omega \in P$ reflects the state of all points as follows: For one point p, $\omega(p) = (r, c, u)$ is a triple denoting the requested state r as set by the controller, the current point state c and the time tick u when the request was issued. A correctly operating point will always manage to perform a state transition from current to requested state within the deadline $[u, u + \delta_p]$. During that time interval U is possible as intermediate state. A point missing the deadline is considered as non-operational and assumes state F.

3.2.3.3. Signal states SIG. The set of signal states is defined as $SIG = Sid \to (ReqSigState \times SigState \times STM)$ with $STM = \mathbb{N}$ and $ReqSigState = \{H, GL, GS, GR\}$. For $\sigma \in SIG$ and signal s, $\sigma(s) = (r, c, u)$ is a triple denoting the requested state r as set by the controller, the current point state c and the time tick u when the request was issued. A correctly operating signal reaches state $r = c$ within deadline $[u, u + \delta_s]$, otherwise it assumes state F. During that time interval U is possible as intermediate state.

3.2.4. Safe states. A state is said to be *safe* if the following invariant $is\_safe(\alpha, \omega, \sigma, t, ctrl)$ holds:

$$
\begin{aligned}
&(\forall\, g_1, g_2 \in G \bullet \eta(g_1, g_2) \Rightarrow \\
&\qquad dst(g_1) = g_2 \Rightarrow dst(g_2) \neq g_1) \\
&\wedge (\forall\, g_1, g_2 \in G \bullet \\
&\qquad dst(g_1) \neq \epsilon \wedge dst(g_1) = dst(g_2) \Rightarrow g_1 = g_2) \\
&\wedge (\forall\, p \in Pid \bullet \mathbf{let}\ g = stem\_sensor(p), \\
&\qquad \{g_2, g_3\} = branch\_sensors(p)\ \mathbf{in} \\
&\qquad |\ ctr(g) - ctr(g_2) - ctr(g_3)\ | \leq max) \\
&\wedge (\forall\, g \in G_0, g_1 \in G \bullet \eta(g, g_1) \Rightarrow \\
&\qquad |\ ctr(g) - ctr(g_1)\ | \leq max) \\
&\wedge (\forall\, p \in Pid \bullet \mathbf{let}\ g = stem\_sensor(p), \\
&\qquad \{g_2, g_3\} = branch\_sensors(p)\ \mathbf{in} \\
&\qquad c = U \Rightarrow ctr(g) = ctr(g_2) + ctr(g_3)) \\
&\wedge (crossing((g_1, g_2), (g_3, g_4)) \Rightarrow \\
&\qquad ctr(g_1) = ctr(g_2) \vee ctr(g_3) = ctr(g_4))
\end{aligned}
$$

$G_0 = G - (\text{ran } stem\_sensor \cup (\bigcup \text{ran } branch\_sensors))$ defines the set of sensors which are not associated with a point.

Informally speaking, $is\_safe(\alpha, \omega, \sigma, t, ctrl)$ expresses the following properties:

(1) No trams are driving in opposite directions on the same segment (conjunct 1),
(2) no trams are moving in opposite direction towards the same sensor (conjunct 2),
(3) the number of trams approaching a point from stem and branches at the same time is at most 1 (conjunct 2),
(4) on each segment the number of trams passing in one direction is less than a predefined maximum *max* (conjuncts 3, 4),
(5) there are no trams on a point which is in an undefined state (U) (conjunct 5), and
(6) there are not trams at the same time on two different segments that cross each other (conjunct 6).

3.2.5. Transition relations. The *NTS* transition relation $T \subseteq S \times S$ is defined as the concurrent composition of transition relations for each sensor, point, signal, and a time ticks described below. In the description `(sens, pt, sig, t, ctrl)` denotes the state vector.

3.2.5.1. Time ticks. We model the passage of time as a discrete increment of the *TM* state component.

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

3.2.5.2. Transition relation for signals. Any signal s has a transition rule stating that when the actual state is different from the requested state and F, the actual state can be changed to state U or the requested state before the deadline is met and to state F after the deadline has passed:

```
let (r, c, u) = sig(s) in
  if ((c != r) and (c != F)
              and (t <= u + delta_s)) {
    sig(s) = (r, U, u);
  } [] if ((c != r) and (c != F)
              and (t <= u + delta_s)) {
    sig(s) = (r, r, t);
  } [] if ((c != r) and (c != F)
              and (t >  u + delta_s)) {
    sig(s) = (r, F, t);
  }
```

3.2.5.3. 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.

3.2.5.4. Transition relation for sensors. The transition relations for sensors are classified according to the sensor locations (1) network boundary (i.e. only one neighbouring sensor), (2) between two neighbouring segments without a point and (3) at stem and branches of a point. Furthermore, the transition rules depend on whether a signal is associated with the sensor or not. Due to the usual space limitations, we only give the transition rule for sensors located at two connecting segments without point. We use Boolean expression nb(g0,g1,g2) to express the neighbourhood relation $\eta(g_0, g_1) \wedge \eta(g_1, g_2) \wedge g_0 \neq g_1 \wedge g_0 \neq g_2 \wedge g_1 \neq g_2$.

```
let (x0, n0, d0, u0) = sens(g0),
    (x1, n1, d1, u1) = sens(g1),
    (x2, n2, d2, u2) = sens(g2) in
if ( nb(g0,g1,g2) and x1 == LO and
     n0 > n1  and t > u1 + delta_c and
     (g1,g2) in dom(signal) and
     sig(signal(g1,g2)) = GS ) {
  sens(g1) = (HI, n1 + 1, g2, t);
}
```

This rule covers the situation where sensor g_1 has been in state LO for at least δ_c 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. The condition $n_0 > n_1$ on counters expresses the fact that a tram is approaching g_1 from g_0; in fact, $d_0 = g_1$ holds. If signal at g_1 in direction g_2 shows aspect GO then sensor g_1 may perform a transition into state HI, meaning *"tram from g_0 to g_1 now passes g_1*. As a consequence, g_1's counter is incremented and g_2 is recorded as the destination sensor of g_1. If g_2 were approached from another direction simultaneously, this would result in an additional sensor having g_2 as destination sensor, so that the second conjunct of the safety conditions would be violated. The same preconditions apply for a transition into intermediate state UHI, but here the sensor state is updated as sens(g1) = (UHI, n1, d1, t).

The destination sensor is removed from g_1 as soon as the counters of neighbouring sensors indicate that no tram is currently passing from g_1 to either of its neighbouring sensors:

```
let (x0, n0, d0, u0) = sens(g0),
    (x1, n1, d1, u1) = sens(g1),
    (x2, n2, d2, u2) = sens(g2) in
if ( nb(g0,g1,g2) and n0 >= n1 and
     n1 <= n2 and d1 != epsilon
     and t > u1 +  delta_c ) {
  sens(g1) = (x1, n1, epsilon, u1);
}
```

4. CONCLUSIONS

We have described an approach for the automated verification of tramway control systems. Conventional development approaches for similar systems verify software and application-specific configuration data in separate verification steps, using different techniques. In particular, software is tested in a non-exhaustive way, so that new errors are often encountered when the software is executed with new configuration data. Conversely, implicit assumptions and rules encoded in the software may be overlooked during the verification of configuration data. Therefore we advocate model checking the complete executable system consisting of executable re-usable interpreter code plus application-specific configuration data. Since the verification is automatic, the additional effort for re-verifying the interpreter code with each new configuration may be neglected.

It is currently investigated which model checking techniques are best-suited for our specific task. We expect that a considerable reduction of the state space can be achieved using the bounded model checking approach described in (Große and Drechsler, 2003): The authors describe the possibility of unfolding the state space for a bounded number of discrete time ticks only. Though it cannot be guaranteed in our context that a tram will traverse the network in bounded time, all trams only need a bounded number of steps (i. e., transitions between track segments) for this traversal. Therefore an abstract time notion may be defined by the number of steps a specific tram has proceeded within the network.

Observe that the configuration of the controller enforcing safety conditions for the physical network model using global control variables may be regarded as a typical *rely-guarantee* situation (de Roever et al., 2001, pp. 363): Two state transition systems interact with each other using shared variables, the controller STS guaranteeing certain safety properties while relying on the physical conditions modelled in the network STS. It is currently investigated whether the rely-guarantee method can simplify the formal verification of the interpreter software.

REFERENCES

de Roever et al., W. P. (2001). *Concurrency Verification.* Cambridge University Press.

Große, D. and R. Drechsler (2003). Formal Verification of LTL Formulas for SystemC Designs. In: *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS'03), Bangkok, 2003.*

Haxthausen, A. E. and J. Peleska (2000). 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 (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). 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. To appear.