GENERATION OF EXECUTABLE RAILWAY CONTROL COMPONENTS FROM DOMAIN-SPECIFIC DESCRIPTIONS

Anne E. Haxthausen[1], **Jan Peleska**[2]

[1] *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
[2] *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: jp@tzi.de

Abstract: In this paper we present a domain-specific description language for railway and tramway control systems and explain how one can generate an executable control system from descriptions written in that language. The control systems are route based interlocking systems and a domain-specific description consists of a description of the network topology, the interlocking tables and the hardware devices. The main idea for generating such systems is to have a generic framework of state transition systems and a compiler that can transform a domain-specific description into configuration data for the generic framework. The resulting state transition system instance provides the semantics of the domain-specific description and is made executable by an interpreter.
Keywords: railway control systems, domain-specific language, formal methods

1. INTRODUCTION

In recent years, the development of domain-specific description languages have gained wide interest, specifically in the domain of railway control systems. In contrast to general wide spectrum specification formalisms, domain-specific descriptions use the terminology of the application domain, thereby facilitating the communication between application experts and the specialists developing computerized control systems.

In this paper, we describe how specifications written in a domain-specific language (Peleska *et al.*, 2000; Haxthausen and Peleska, 2002) for railways and tramways (focusing on the latter) can be transformed into executable control components. The possibility to generate executable control systems directly from the specification considerably increases the "return of investment" to be gained from the specification effort.

Our concept is based on four models: (1) The *(tramway or railway) network model* describes the static, physical aspects of the track network the trains or trams are moving on. This model can be enhanced with dynamic aspects to a model with a well-defined Labelled Transition System (LTS) semantics. In particular, safe and unsafe situations can be precisely specified, as in (Haxthausen and Peleska, 2000). (2) The *hardware model* specifies the types of track elements (sensors, signals, points, railway crossings) to be used. This is required because the driver selection and driver interfaces may depend on the specific type of sensor, signal etc. In addition, the hardware model describes the target controller hardware where the executable software system has to be allocated. The most important aspect of this description is the number of available CPUs (two or more CPUs allow for a parallelisation of control algorithms, but it has to be ensured that the parallelisation is safe) and memory maps (where code and data are to be allocated). The hardware model has a well-defined syntax (table format), but we do not associate a formal semantics. (3) The *route model* consists of tables for the specified routes to be used by trains or trams within the network and the associated point positions and signal states. As in the hardware model, a table syntax is specified. In addition, it has a static semantics relating specified routes and their associated track elements to the network model. A behavioral semantics is associated implicitly by means of a transformation into the fourth model: (4) The *execution model* consists of programming language patterns instantiated by configuration data. The configuration data encodes the information about routes, points and signals obtained from route model and hardware model by means of transformation. The instantiated patterns define possible transitions of a state transition system (STS). Hence, the instantiated model has a well-defined state transition systems semantics. Intuitively speaking, the model specifies the possible switching sequences for signals and points which allow trains or trams to traverse the network along the intended routes. The instantiated execution model can be executed directly by means of reusable interpreters that select certain executions of the associated STS.

It will turn out in the subsequent sections that our modelling techniques are – when compared to wide spectrum (semi-) formal specification languages – fairly simple. According to the authors' point of view, this simplicity is an essential prerequisite for the feasibility of completely automatic transformations into executable systems and for trustworthy and comprehensive safety verifications. We have tried to use the simplest modelling techniques with sufficient expressive power to specify all safety-related aspects and all behavioural aspects of the executable system.

First, in *Section 2*, we introduce the technical concept of the control systems we generate. *Section 4* sketches the verification, validation and test concept which is elaborated in more detail in another paper (Haxthausen and Peleska, 2003). In *Section 3* we present the domain-specific language (the network model and route model), and in *Section 5* we outline the semantics of the language (the execution model) and explain the principles for generating an executable system from a description in the language. Finally, in *Section 6*, we describe related work.

2. ENGINEERING CONCEPT

In this section, we introduce the technical concept of the control systems we generate. Like in (Lindegaard *et al.*, 2000), we have taken a routed based approach to control.

Every control system within the domain will interact with external hardware sub-systems: *Trams* move through the tramway network on *routes* which have been pre-defined in the specification of the control application. The position of trams is detected by the control system via *sensors*. If trams are equipped with on-board *tram control computers*, these may send *route requests* to the control system. Otherwise the control system will determine the route to be reserved for the tram based on the sensor information, additional timetable information or manual interactions with the control system. According to the actual state of route requests and tram locations within the network, the controller will switch *points* in order to establish the requested routes, as long as these do not conflict with routes currently allocated by other trams. After proper point positions have been ensured along the route, conflicting routes have been blocked via *signals* and no trams use parts of the route in a way that conflicts with the safety requirements, the control system sets the entry signal of the route into state GO (straight, left, right) to indicate that the tram may enter the route. If tram control computers are available, the go-signal is also sent to the tram itself, so that external signals become superfluous if all trams are equipped with reliable tram control computers. When a tram has entered the route, the control system sets the entry *signal* of the route to HALT.

The safety control layer of a control system has the following control components:

- A *safety monitor* that performs cyclic checks that all track elements operate correctly and that no two trams are on conflicting routes at the same time.
- A *route dispatcher* that registers route requests and initiates the allocation of routes in compliance with the safety requirements and scheduling requirements.
- For each route, a *route controller* that is responsible for setting the points and entry signal of the route during allocation and deallocation of the route.

3. DOMAIN-SPECIFIC DESCRIPTIONS

A domain-specific language (Peleska *et al.*, 2000; Haxthausen and Peleska, 2002) for tramway/railway control systems has been designed. A description in this language consists of a *network* description, a set of *interlocking tables* and a *hardware* description. These descriptions possess both textual and graphical representations. Below we describe the two first kinds of descriptions.

3.1 Network descriptions

A network description represents the topology of the track elements, their positions and interconnections.

An example of a graphical representation of such a description is shown in *Figure 1*. In this example the network consists of the following track elements:

- controllable *points*: W100, W102, W118
- *signals*: S20, S21, S22
- *sensors*: G20.0,...,G25.1
- *track segments*, shown as solid lines between two sensors, e.g. [G20.0,G20.1]

3.2 Interlocking tables

The interlocking tables comprise four tables: (1) a *route definition table* that describes the possible routes through the network, (2) a *route conflict table* that describes which routes must not be allocated at the same time (because they overlap some how), (3) a *point position table* that describes for each route how points should be set for the route to be traversable, and (4) a *signal setting table* that describes for each route the name of its entry signal and the aspect it should be set to in order to signal that a tram is allowed to enter the route. Below is shown a graphical representation of these tables for the tramway network in *Figure 1*. More detailed explanations of the tables can be found in (Haxthausen and Peleska, 2002).

The static semantics of the tables is a predicate checking that the tables are well-formed wrt. the network description. It checks for instance that a signal for a route in the signal setting table really is physically placed at the entry of the route in the network.

Route definition table	
Route	**Route Sensor Sequence**
1	$\langle G20.1, G20.2, G21.0, G21.1\rangle$
2	$\langle G20.1, G20.3, G25.0, G25.1\rangle$
3	$\langle G22.1, G22.2, G23.0, G23.1\rangle$
4	$\langle G22.1, G22.3, G25.0, G25.1\rangle$
5	$\langle G24.1, G24.3, G23.0, G23.1\rangle$
6	$\langle G24.1, G24.2, G21.0, G21.1\rangle$

Route conflict table						
Route	**Conflicts with**					
	1	2	3	4	5	6
1		•				◦
2	•		◦	◦		◦
3		◦		•	◦	◦
4		◦	•			
5			◦			•
6	◦	◦	◦		•	

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

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

4. DEVELOPMENT, VERIFICATION, VALIDATION AND TEST CONCEPT

In this section we sketch the interaction between processes for development, verification, validation and test (VVT).

Development starts with a network model and an informal description of routes as inputs. The first development step then consists in the creation of the route model as described in *Section 3* and checking that it is statically well-formed Next, the hardware model is elaborated: The track elements are listed, together with their hardware types required for driver selection and interface interpretation. The available controller components and CPUs are specified.

An automatic transformation compiles the route model and the hardware model into configuration data for the execution model. The execution is performed by reusable interpreters which are formally verified once and for all to perform according to the semantics of a state-transition system (see *Section 5*), if given correct configuration data. Moreover, since an abstraction relation between the STS associated with the configuration data and the LTS associated with the enhanced, dynamical network model has been elaborated (Haxthausen and Peleska, 2003), we can verify the execution model instantiated with the concrete configuration data against the enhanced network model by means of model checking. Since this check can be performed exhaustively, it is not required to verify the transformation from route model to execution model: Any transformation error

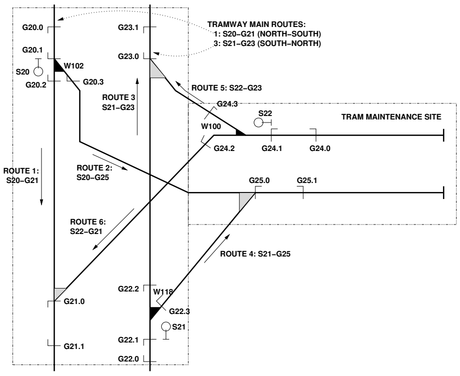

Fig. 1. Network description.

would be revealed in the model checking process. Moreover, any specification error which occurred in the manual process of route model development will be revealed as a violation of safety or other constraints specified for the network model.

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 model specifies memory boundaries, available processors and interface types. 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 (Peleska, 2002).

5. SEMANTICS OF EXECUTABLE COMPONENTS

In this section we will introduce the concept of state transitions systems (STS) and define a restrictive pattern of programming language commands which automatically ensures that the program will only perform executions of an associated STS.

5.1 State transition systems and restricted programming patterns

A *state transition system* $STS = (S, s_0, T)$ consists of a set S of states, an initial state s_0 and a transition relation $T \subseteq S \times S$. The system STS is called *deterministic* if T is a partial function $T : S \nrightarrow S$, that is, if each state s has at most one post state $T(s)$. An *execution* of STS is a sequence $s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow \ldots$ of state transitions such that $(s_i, s_{i+1}) \in T$ for $i = 0, 1, 2, \ldots$. For two STS over the same state space and with the same initial state, say $STS_1 = (S, s_0, T_1)$ and $STS_2 = (S, s_0, T_2)$, we say that STS_2 *refines* STS_1 if $T_2 \subseteq T_1$ holds, and therefore, the executions of STS_2 build a subset of STS_1 executions.

For the specification of states it is useful to model S as a collection of valuation functions $s : V \rightarrow D$, where V is a set of variable symbols and each state s is a function associating the variable symbol with its current value in domain D. State changes may now be expressed by functional overriding: $s_1 \longrightarrow s_2$ is specified as $s_2 = s_1 \oplus \{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$. If states are modelled as variable valuations, they can be identified via by their *variable values*: A predicate $p(x_1, \ldots, x_k)$ with free variables $\{x_1, \ldots, x_k\} \subseteq V$ and n-ary functions over D^n identifies the states $\{\, s \in S \mid p(s(x_1), \ldots, s(x_k)) \,\}$. Using a programming language syntax, transition relations may now be specified as

```
if ( p(x1,...,xk) ) {
  xn1 = expr1; xn2 = expr2; ...
}
```

which has the meaning (we skip the usual explanation of expressions over variable symbols, constants and operators defined on D):

$$
\begin{array}{l}
\{s \longrightarrow s' \mid \\
\quad p(s(x_1), \ldots, s(x_k)) \wedge \\
\quad s' = (\ldots((s \\
\qquad\qquad \oplus \{x_{n_1} \mapsto \mathit{expr}_1\}) \\
\qquad\qquad \oplus \{x_{n_2} \mapsto \mathit{expr}_2\}) \ldots) \\
\} \\
\subseteq T
\end{array}
$$

Observe the position of brackets in the $\oplus$-expression; it implies that `{ xn1 = expr1; xn2 = expr2; ... }` is interpreted in the usual program assignment style: Every occurrence of `xn1` in `expr2` has to be substituted by the value of `expr1` of the most recent assignment to `xn1`.

A state transition system is deterministic, if an ordering of transition rules exists such that they may be expressed as

```
if ( p1 ) { assignments1 }
if ( not (p1) and p2 )
   { assignments2 }
...
if ( not (p1 or ... or pn)
     and pn+1 )
   { assignmentsn+1 }
```

For this situation, the simpler notation

```
if ( p1 ) { assignments1 }
else if ( p2 ) { assignments2 }
...
else if ( pn+1 ) { assignmentsn+1 }
```

may be used. For a given $STS_1 = (S, s_0, T_1)$ we can define deterministic refinements $STS_2 = (S, s_0, T_2)$ by introducing an ordering `r1,...rm` of the transition rules from T_1 and defining T_2 by the rule `r1 else r2 else ...rm`.

If $STS_1 = (S, s_0, T_1)$ and $STS_2 = (S, s_0, T_2)$ are two state transition systems over the same state space S and with identical initial state s_0, we define their parallel combination $(STS_1 \| STS_2) = (S, s_0, (T_1 \mid T_2))$, where the set of transitions $(T_1 \mid T_2)$ is the union of transitions from T_1 and T_2, and transitions

```
if (p1(x1,...,xk) and p2(x1,...,xk))
  { xn1 = expr1; xn2 = expr2;... }
```

where
`xn1 = expr1; xn2 = expr2;...`
is an interleaving of
`xn11 = expr11; xn12 = expr12;...`
and
`xn21 = expr21; xn22 = expr22;...`
and

```
if ( pi(x1,...,xk) ) {
    xni1 = expri1;
    xni2 = expri2;
    ...
}
```

is a transition of T_i, for $i = 1, 2$.

Intuitively speaking, parallel composition admits any asynchronous interleaving of read or write operations from STS_1 and STS_2 which would also be possible in shared variable programs executing on two distinct processors without synchronisation mechanisms. STS_1 may be "faster" than STS_2 and vice versa: Several STS_1-transitions may be performed before one STS_2 transition or one joint transition occurs. By

```
STS_i() {
 while ( condition ) {
  if ( p1 ) { assignments1 }
  else if ( p2 ) { assignments2 }
  ...
  else if ( pn+1 ) { assignmentsn+1 }
 }
}
```

we introduce another notation for the executions of deterministic STS_i. With condition `true` we specify the executions of unbounded length. Observe that deterministic STS_1 only has one unbounded length of execution while $(STS_1 \| STS_2)$ may have many. If STS_i, $i = 1, 2$, are deterministic STS then a specific subset of executions from $(STS_1 \| STS_2)$ can be generated by applying the transitions rules of STS_1 and STS_2 (which are also valid rules of $(STS_1 \| STS_2)$!) in strict alternation. For this type of executions we introduce syntax

```
while ( condition )
  { block_1; block_2 }
```

where `block_i` denotes the blocks inside the `while`-loops of STS_i, $i = 1, 2$.

We have now introduced a restrictive syntax pattern for while-programming languages which ensures that the program will always perform executions of the associated STS specified by the program variable state space and the transition rules given by `if-else` expressions inside the `while` block. Parallel execution of these programs on several processors or alternating executions of the transition blocks of each program results in valid executions of the parallel combination of the associated STS. This equivalence between programs and STS executions holds, provided that

1. The compiler transforms `while` and `if-else` control structures correctly according our semantic STS interpretation,

2. The compiled conditions evaluate in a way which is consistent with our semantic condition interpretation (evaluation of predicates in domain D),

3. The assignments are implemented in accordance with our functional overriding interpretation.

The first condition should be fulfilled by any reasonable compiler – if in doubt, the control structures can be coded directly in assembler. The second condition would be non-trivial to verify if we admitted arbitrary condition expressions. Fortunately, this is not the case for our domain-specific application: It will be illustrated in the example below that all condition expressions we require are derived from the pattern

$$\bigwedge_{i=0}^{k-1}(x_i \& m_i)\rho(y_i \& m_i)$$

where x_i is an unsigned integer variable, y_i is an unsigned integer variable or a constant value, m_i is a constant, & is the bitwise AND operation and ρ is one of the relational operators $<, \leq, >, \geq, \neq, ==$. All expressions used in assignments associated with transitions are derived from the pattern

```
(x & (~m)) | (((x & m) op y) & m);
```

where the result of the expression is assigned to an unsigned integer variable x, y is a variable or a constant, m is a constant, *op* is one of the operators `+`,`-`,`|`,`&`, and `|` denotes bitwise OR and `~` denotes bitwise negation. This type of expressions changes x at the bits masked by m, where the result from operation x *op* y becomes effective. These expression patterns for conditions and assignments can be easily coded in assembler, and the assembler code can be formally verified based on the machine model of the underlying CPU. Because of the restrictive use of control structures and expressions, we believe that a complete formal verification is feasible. This task is presently started in the authors' research groups.

For the executions of a (possibly) nondeterministic STS (not to be expressed in a conventional programming language), we use notation

```
STS() {
 while ( condition ) {
  if ( p1 ) { assignments1 }
  []
  if ( p2 ) { assignments2 }
  []
  ...
  []
  if ( pn ) { assignments_n }
 }
}
```

to specify its subset of executions restricted by `condition`.

5.2 Implementation considerations

The considerations about STS and programming languages described above motivate an implementation as a purely state-based system without any interrupts, signals or event handling. This approach induces the following *hardware abstraction* concepts (*Figure 2*). Observe that these concepts are available today and applied by various companies operating in the railway domain.

The driver software resides on I/O boards with local processors allowing to perform low-level driver tasks like signal debouncing and data format transformations locally on the interface board without CPU participation. The interface between I/O boards and safety control layer (i.e. the driver interface) consists of a *dual ported RAM*: The CPU writes/reads interface data via interface bus to/from a memory section of the I/O board. The driver also places input data into this memory area. Memory access can be performed without utilisation of interrupts; so the safety control layer polls for new input data and the on-board drivers poll for new commands from the safety control layer. For all track elements we define a general interface pattern: The CPU writes the requested element state (e.g. signal state HALT) into a dedicated request area of the I/O board memory and sets the *switching deadline*, i.e., the latest admissible point in time when the requested state must be reached. The driver writes the actual element state (e. g. signal state GO) into the *actual state* memory area. Requested and actual states are uniformly encoded as specific bit patterns on unsigned integers. Using this concept, the *local invariant safety conditions* for each signal or point may be specified according to the uniform pattern

$$\texttt{sw\_deadline} < \texttt{timetick} \Rightarrow (\texttt{req\_state\&mask}) == (\texttt{act\_state\&mask})$$

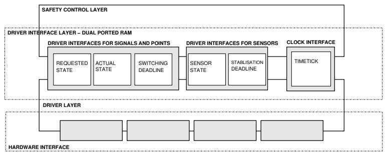

Fig. 2. Hardware abstraction at the interface between drivers and safety control layer.

Sensor states are uniformly encoded as `HIGH, LOW, UNSTABLE`. When a change in the sensor signal occurs, the driver sets its state to `UNSTABLE` and defines a *stabilisation deadline* which is the latest point in time when the debouncing phase must have reached a stable new sensor state. The resulting safety condition is

$$\texttt{stab\_deadline} < \texttt{timetick} \Rightarrow \texttt{sensor\_state} \in \{\texttt{HIGH,LOW}\}$$

5.3 Examples

Using the STS concept introduced above, the full control system can be modelled as the parallel combination

$$(SM\|D\|RC_1\|\ldots\|RC_n\| DRV_1\|\ldots\|DRV_k\|CLOCK)$$

with safety monitor SM, route dispatcher D, route controllers RC_i, drivers DRV_i and time tick generator $CLOCK$.

The visible behaviour on the driver interface appears to be nondeterministic, since it depends on the state and behavior of the physical track elements. Therefore the possible executions of each driver are instantiated from the pattern

```
DRV() {
  while(true) {
   if ( true ) actual_state = v1;
   []
   ...
   []
   if ( true ) actual_state = vn;
  }
}
```

where `v1,...,vn` are all possible values for the associated track element. The set of executions representing normal (i.e., correct) behavior of the driver – this corresponds to correctly operating track elements, interface hardware and driver software/firmware – is given by the following pattern:

```
DRV() {
 while(true) {
  if (requested_state
          != actual_state
      and
      timetick = sw_deadline
     ) {
      actual_state = requested_state;
  }
  else {
   if (requested_state !=
        actual_state) {
      actual_state = requested_state;
   }
   []
   if ( true ) ;
  }
 }
```

Correctly operating drivers and track elements will guarantee that actual state equals requested state by the time the deadline has been reached. For the verification of the safety control layer, these different behaviours are generated by the model checker in order to verify that the control layer reacts properly in all situations. For the verification, the pattern has to be instantiated for each track element and its associated driver.

As an example from the safety control layer, consider the part of the safety monitor supervising the correct track element status for all signals and points. This portion of *SM* can be instantiated from the execution pattern

```
SM() {
 while (true) {
  for (i = 0; i < NUM_ELEMENTS; i++ ) {
    if (requested_state[i] !=
          actual_state[i]
        and timetick > sw_deadline[i]) {
      //transition to stable safe state
      //set all signals to HALT ...
      for (j = first_signal_idx;
           j <= last_signal_idx; j++) {
         requested_state[j] = HALT_CMD[j];
         sw_deadline[j] = timetick + DELTA;
      }
    }
  }
 }
}
```

The safety monitor checks for each element whether the safety violation *actual state differs from requested state and deadline has been reached* has occurred. If this is the case, it sets the requested states of all signals to halt (in the real system, additional activities are performed not shown here). The safety monitor could also run in parallel instances; we have chosen the serialised version as discussed above. The specification of possible *SM* executions is generic in the number (`NUM_ELEMENTS`) of track elements, the number of signals (`first_signal_idx` and `last_signal_idx`) and the bit pattern for the HALT command. During transformation process these constant data structures will be generated and the state variables (such as `requested_state[j]`) will be defined with appropriate size derived from the hardware model.

6. RELATED WORK

The work presented in this paper focuses on the description of the execution model, the transformation from route model into configuration data and the interpreter-based execution of the instantiated model. In another paper currently under preparation (Haxthausen and Peleska, 2003), the authors describe the automatic generation of verification, validation and test suites derived from domain-specific descriptions for the same type of systems considered here and in (Peleska *et al.*, 2000; Haxthausen and Peleska, 2002; Haxthausen and Peleska, 2000): The tram movements allowed by the execution model instantiated with the configuration data are checked to be safe within the context of the network model.

Our elaborations are based on previous publications (Peleska *et al.*, 2000; Haxthausen and Peleska, 2002; Haxthausen and Peleska, 2000) and performed within the context of the HYBRIS (Efficient Specification of Hybrid Systems) project supported by the Deutsche Forschungsgemeinschaft DFG as part of the priority programme on *Software Specification – Integration of Software Specification Techniques for Applications in Engineering* (DFG,).

Other approaches with domain-specific formalisms in the field of railway control systems have been based on semi-formal description techniques already in use and associated rigorous semantic interpretations with these formalisms, see (Fokkink *et al.*, 1998) as an example.

REFERENCES

DFG, Deutsche Forschungsgemeinschaft. Priority Programme on *Software Specification – Integration of Software Specification Techniques for Applications in Engineering.* Information available under http://tfs.cs.tu-berlin.de/projekte/indspec/SPP.

Fokkink, W. J., G. P. Kolk and S. F. M. van Vlijmen (1998). EURIS, a specification method for distributed interlockings. In: *Proceedings of SAFECOMP '98*. Vol. 1516 of *Lecture Notes in Computer Science*. Springer-Verlag. pp. 296–305.

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). Automatic Verification, Validation and Test for Railway Control Systems based on Domain-Specific Descriptions. Submitted to the 10th IFAC Symposium on Control in Transportation Systems.

Lindegaard, M. P., P. Viuf and A. E. Haxthausen (2000). Modelling Railway Interlocking Systems. In: *Proceedings of the 9th IFAC Symposium on Control in Transportation Systems 2000, June 13-15, 2000, Braunschweig, Germany.* pp. 211–217.

Peleska, J. (2002). Formal Methods for Test Automation - Hard Real-Time Testing of Controllers for the Airbus Aircraft Family. In: *Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology, (IDPT2002), Pasadena, California.*

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.