Integrated Design and Process Technology, **IDPT-2002**
Printed in the United States of America, June, 2002

A domain specific language for railway control systems

Anne E. Haxthausen[1] and Jan Peleska[2,3]
[1] **CSE/IMM, Technical University of Denmark, Building 321, DK-2800 Lyngby, Denmark, e-mail: ah@imm.dtu.dk**
[2] **TZI, Universität Bremen, P.O. Box 330440, D-28334 Bremen, e-mail: jp@tzi.de**
[3] **Verified Systems International GmbH, Bremen**

ABSTRACT: In this paper we present a domain specific language for railway and tramway control systems. In this language network topology, interlocking tables etc. can be described. Adopting terms and concepts from the application domain, this language is well suited for communication between users, system engineers and control computer specialists. The semantics of the language is given in the following way. There is a re-usable framework of generic state machines describing a generic architecture of the considered class of control systems. Descriptions in the domain-specific language are transformed into configuration data. The semantics is then the generic framework instantiated by this configuration data. This approach allows to develop a safety proof using quantification over all admissible sets of configuration data. Such a proof represents a generic theory and would not have to be re-done as long as the configuration data remains admissible. From the formal semantics an executable software system can easily be derived. It is indicated how the integrated hardware/software system can be tested automatically, based on the domain specific description.

I. Introduction

A. Objectives

A stimulating discussion held in both software engineering and systems engineering communities is about so-called *wide spectrum specification languages* versus *domain-specific description formalisms*. The former use language elements which are independent of a specific application context, while the latter utilize terms and concepts of a well-defined application domain. While it is an advantage of wide spectrum languages that a large variety of systems may be specified without changing the formalism, it is often criticized that the meaning of these specifications is hard to understand for people outside the computer-science domain. As a consequence it is questionable whether wide spectrum languages such as UML will ever serve as a useful means of communication between system engineers who are experts in the application domain and computer science specialists handling the software-related control aspects of a system. In contrast to this, domain-specific description formalisms are more intuitive for the application domain, but – just as with wide spectrum languages – their semantics has to be established in a rigorous way, because otherwise it will not be possible to argue about specification correctness in an unambiguous way and derive executable systems which can be proven to be consistent with the specification. As a consequence, this effort is only invested in cases where the application domain is well understood and the criticality of applications implies that formal reasoning about specifications and implementations is mandatory.

As a contribution to this discussion, we introduce a domain-specific description formalism for railway and tramway control systems (focusing on the latter) in this article. The semantics of the description is defined by means of a compilation process into a formal language with well-defined semantics. In addition to associating precise meaning to the domain-specific descriptions, this formal intermediate language is sufficiently close to programming languages, so that the transformation into executable code is straight forward. To facilitate this transformation even more, we use generic re-usable interpreters which are instantiated by the application-specific data defined in the domain specification.

B. Overview

In Section II, the type of control systems captured by our framework is described. A re-usable architecture and the most important abstract classes to be instantiated for a concrete control system are introduced. Section III introduces the domain-specific language elements. In Section IV, the transformation from the domain-specific description into an intermediate formal language is sketched. Section V contains the conclusion, refers to related work and indicates how the verification and validation process for both domain specifications and the resulting executable system is performed.

II. Engineering concept and Re-usable Architecture

Recall that *frameworks*, as defined in [4] are generic collections of cooperating classes associated with a re-usable design structure. By means of instantiation, sub-classing and composition of class instances, frameworks are customized for a specific application context. In this section, we will first introduce the architectural aspects of our tramway control framework and explain the underlying engineering concept.

A. Devices Interacting With the Control System

Every control system within the domain will interact with the following types of external hardware sub-systems, for which we will use the general term *devices* in this article: *Trams* move through the tramway network on *routes* which have been pre-defined in the specification of the control application. The details of this route description will be elaborated in the next section. 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 time-table 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 used by other trams. After proper point positions have been ensured along the route and no trams use parts of the route in a way that conflicts with the safety requirements, the control system switches the *signals* associated with the requested route to indicate that the tram may pass. 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. Trams traveling along established routes are tracked by the control system by recording the sensor signals indicating the present tram location.

B. Generic Architecture

In practice, a tramway network will consist of different types of devices fulfilling the same objectives: For example,

- Signals may be equipped with conventional light bulbs or LED technique.
- The signals of simple sensors have to be de-bounced within the control system, while more elaborate digital sensors transmit signals that can be directly used as valid position information.
- At points in the tramway network, where branching into three directions (left, straight, right) is possible, this may be implemented by two conventional point motors which have to be switched explicitly into (left, straight) and (straight,right) position, respectively, or by a single motor controlling the three directions (left, straight, right) simultaneously.

On the level of the decision routines for establishing routes and checking safety conditions, these technical details are without relevance. Therefore, we propose to use the generic layered architecture shown in Figure 1.

The *hardware interface layer* consists of the concrete device drivers to be used for different types of signals, points, sensors and tram communication systems. Typically, this layer will consist of hardware-specific code which is reusable, as long as the properties of the devices controlled by

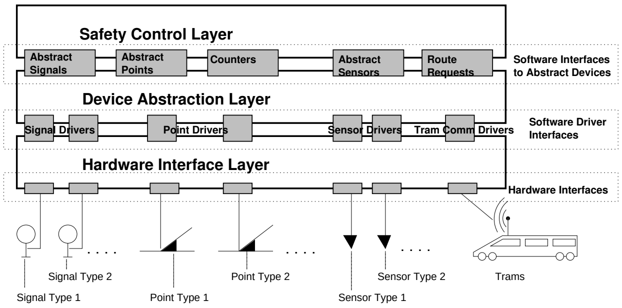

Fig. 1. Re-usable architecture of the Tramway Control Framework.

the driver software do not change. The specific properties of the device are still visible at the *software driver interface*.

The objective of the *device abstraction layer* is to hide the device-specific information from the higher layer by abstracting information obtained from the hardware interface layer to its logical meaning. Conversely, abstract control commands received from the higher layer are transformed into concrete device-dependent commands to be passed to the software driver interfaces. Moreover, errors which do not impair the system safety are handled within the device abstraction layer. The abstraction offered to the higher layer consists of five abstract classes:

- Instances of the *Abstract Signals* class have states (1) F0 = STOP, (2) F1 = GO-STRAIGHT, (3) F2 = GO-LEFT, (4) F3 = GO-RIGHT, (5) F6 = REQUEST-RECEIVED, (6) SIGNAL-OK/FAILURE. F6 is used to signal to the trams that their requests have been received and are currently being processed. SIGNAL-FAILURE indicates that a signal failure impairing the safety-related functionality of the signal has occurred. They offer operations SET-F0,…, SET-F6 to the safety control layer.
- Each object of class *Abstract Points* may have states (1) STRAIGHT-POSITION, (2) LEFT-POSITION, (3) RIGHT-POSITION, (4) POINT-OK/FAILURE. The class defines operations SET-STRAIGHT/LEFT/RIGHT-POSITION.
- Instances of class *Abstract Sensors* offer event information TRAM-PASSES-SENSOR and state information SENSOR-OK/FAILURE.
- *Route Requests* offer an event information ROUTE-<id>-REQUESTED.
- Instances of class *Counters* are associated with certain sensors along the routes and count the number of trams which have passed the sensor. Each counter offers operations INCREMENT-COUNTER and RESET-COUNTER.

It will be shown in the sections below that all global safety properties about the tramway network may be expressed as predicates on counter instances.

The *safety control layer* maintains the global state information about all devices (including trams and their routes) within the network. Based on the abstracted information obtained from the device abstraction layer, all control decisions are taken within the safety layer and passed to the lower layer as abstract control commands for signals and points.

Interpreted in the context of object orientation and frameworks, the device abstraction layer performs sub-classing for the abstract classes whose interfaces are accessed in the safety control layer.

III. Domain-Specific Description Language

The example used to introduce the description language for the tramway control system domain is taken from [12]. It has been modified and enhanced for the present article by additional specification items allowing to generate an executable software system directly from the specification.

A. Tramway Network Specification.

The first component of a domain-specific description for railway control systems consists of the *network*, as displayed for our example in Fig. 2. It represents the topology of the network *elements*, their positions and interconnections. The elements are given by

- *track segments*, each delimited by some neighboring sensors [Gxy.x,Gwv.u] – so, for example, [G20.0,G20.1] is one segment, [G20.3,G25.0] represents another one,
- controllable *points* – W100, W102, W118 in this example,
- *signals* – S20, S21, S22 in this example,
- *sensors* for the detection of passing trams – G20.0,…,G25.1 in the example.

Typically, the network specification will be kept in a CAD system or any other type of database with visualization properties that are suitable to display networks. The symbols used in the visualization are the familiar ones for experts in the railway or tramway domain. The database will store the network specification in such a way that the various elements of the network and the topology can be distinguished and retrieved.

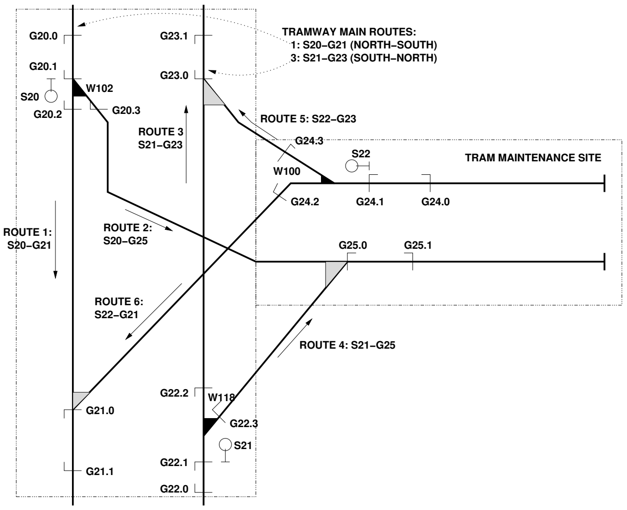

Fig. 2. Tramway network.

B. Interlocking Tables

The network specification is complemented by a set of *interlocking tables*, whose purpose it is to fix the possible routes through the network and the safety conditions which are sufficient to allow trams to pass along these routes.

B.1 Route Definition Table

The first interlocking table is the *route definition table* shown below. For each route, it specifies the sequence of sensors which have to be passed by a tram following this route. For example, route 5 specified in the table, starts at the sensor G24.0, has to pass G24.1 and point W100 in direction W100:RIGHT in order to reach sensors G24.3, G23.0, and is finished when the tram reaches sensor G23.1. The conditions for when a route is requested are also specified. For example, route 5 is requested when a tram located at G24.0 requests for going in direction W100:RIGHT.

Route definition table

Route	Route Sensor Sequence	Request Condition
1	$\langle G20.1, G20.2, G21.0, G21.1\rangle$	G20.0 ∧ W102:STRAIGHT
2	$\langle G20.1, G20.3, G25.0, G25.1\rangle$	G20.0 ∧ W102:LEFT
3	$\langle G22.1, G22.2, G23.0, G23.1\rangle$	G22.0 ∧ W118:STRAIGHT
4	$\langle G22.1, G22.3, G25.0, G25.1\rangle$	G22.0 ∧ W118:RIGHT
5	$\langle G24.1, G24.3, G23.0, G23.1\rangle$	G24.0 ∧ W100:RIGHT
6	$\langle G24.1, G24.2, G21.0, G21.1\rangle$	G24.0 ∧ W100:STRAIGHT

B.2 Route Conflict Table

For a tram to pass along a specific route, it is necessary to *allocate* the route beforehand. Allocation of a route R involves three activities: First, it must be ensured that no conflicting route is already allocated, second, the points along R must be switched and locked into their proper positions, and finally the entry signal of R must be set on go in the direction of the route. For the first of these activities, the *route conflict table* lists the mutual exclusion conditions for different routes. For example, route 1 may only be allocated to a tram if neither route 2 nor route 6 are allocated at the time route 1 is requested. The exclusion is either ensured by point positions making it impossible for R and the conflicting route to be allocated at the same time (this is marked by "•" in the route conflict table), or by signals switched into halt positions, thereby preventing trams to enter the conflicting route (marked by "○"). Observe that the route conflict table is an auxiliary table which could also be systematically derived from the network specification and the route definition table. However, the conflict table is essential for safety considerations, because it specifies conditions which are necessary to avoid collisions. Therefore it is presented explicitly.

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

B.3 Point Position Table

Before a route can be taken, the related points have to be switched into the appropriate position and locked, so that they may not accidentally switched to an other position while a tram is still on the route. The *point position table* below specifies for each point which is part of the route whether it has to be locked in straight, left or right position. If the entry is marked as "—", this means that the point position is not relevant for the route under consideration.

Point position table			
Route	**W100**	**W102**	**W118**
1	—	STRAIGHT	—
2	—	LEFT	—
3	—	—	STRAIGHT
4	—	—	RIGHT
5	RIGHT	—	—
6	STRAIGHT	—	—

Readers who are familiar with train control systems may have noticed that the point position table does not guarantee *flank protection*: Points which are not part of a route may be switched into any position. This is allowed for tramway control systems since – at least in Germany – trams drive according to the road traffic regulations, so that the more strict railway regulations concerning flank protection do not apply.

B.4 Signal Setting Table

At the end of the allocation of a route, the entry signal of the route should be set in accordance with the direction of the route. The *signal setting table* below specifies for each route the name of its entry signal and the state (F1, F2 or F3) it should be set to.

Signal setting table	
Route	**Signal Setting**
1	S20:F1
2	S20:F2
3	S21:F1
4	S21:F3
5	S22:F3
6	S22:F1

IV. Control Aspects of the Tramway Control Framework

A. Components of the Safety Control Layer

The safety control layer introduced above is associated with the following control components:

- The *safety monitor* performs cyclic checks of all safety conditions derived from the interlocking tables.
- The *route dispatcher* registers route requests and initiates the allocation of routes in compliance with the safety requirements and scheduling requirements (such as user-defined priorities or time tables which enhance the domain specification introduced above).
- For each route, a *route controller* is instantiated. Each of these instances is responsible for setting the points and signals during route allocation and maintaining counters to determine when a route is free again.

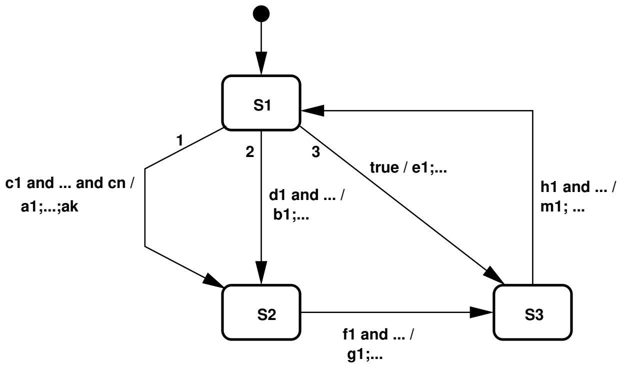

Fig. 3. Sketch of a deterministic sequential state machine.

Each of the control components listed above is an instance of a generic class of *deterministic sequential state machines* which are sketched in Figure 3. Different transitions emanating from a state are prioritized by numbers indicating the order in which the trigger conditions for the different transitions should be evaluated. This enforces a deterministic behavior as expressed by a

```
if (conditions for transition 1 hold)
then perform transition 1;
else if (conditions for transition 2 hold)
then perform transition 2;
...
else if (conditions for transition
        with highest priority value hold)
then perform this transition;
```

```
else remain in present state;
```

statement. The trigger condition of a transaction consists of a *condition list*, that is, a conjunction of *atomic conditions*. An atomic condition has one of the forms $(objId_1, \rho, objId_2)$ or $(objId_1, \rho, const)$, where $objId_i$ are references to objects (abstract signals, points, counters, sensors, route requests), ρ is a binary relation and $const$ a constant value. The conditions evaluate to $true$ if and only if the states s_1, s_2 of the referenced objects are related by $\rho(s_1, s_2)$ or if $\rho(s_1, const)$ holds, respectively. If $\vee$-conditions should trigger a transition, the condition list is first transformed into a disjunctive form and then split into several transitions with the same source and target state, one transition for each disjunct. If a transition fires, the associated *action list* is executed and the target state associated with the transition is taken. The action list consists of a possibly empty list of *atomic actions*, executed sequentially in the order defined by the list. Each atomic action is specified by triples $(objId_1, \omega, objId_2)$ or $(objId_1, \omega, const)$, where ω is a binary operation. The former triple has interpretation "Change the state of object $objId_1$ to new value $\omega(s_1, s_2)$", s_i denoting the pre-states of the referenced objects. The latter has interpretation "Change the state of object $objId_1$ to new value $\omega(s_1, const)$", where $const$ is again a constant value. If none of the trigger conditions apply, the state machine remains in its present state.

Observe that for signals and points the actions set the *requested* object state while the device abstraction layer updates the *actual* object state which is evaluated in the conditions.

The instantiation of each concrete state machine is performed by deriving the number of states, associated transitions and labels from the interlocking tables. This will be illustrated by a closer investigation of the state machine for the safety monitor.

For the route dispatcher priorities on the state transitions can be derived from a priority description that enhances the domain specification introduced above. The description could for instance specify that allocation of the (main) routes 1 and 3 has higher priority than allocation of the other routes. These priorities are used by the route dispatcher in case of conflicting route requests.

B. Configuration Data for the Safety Monitor State Machine

The safety monitor has to check all safety-related conditions applicable in each state of the control system. If a safety condition is violated, the monitor triggers a transition into stable safe state. In our example, this means that all signals must be switched into state F0 until the system is reset manually. This requirement is automatically derived from the conflict table: Whenever a safety violation occurs on a route r then a safety strategy is to block all routes contained in the transitive closure of the conflict relation starting with r as defined in the conflict table. Only routes with no relation to r and any of its conflicting routes may still be used. In the example used in this paper, the transitive closure under the conflict relation starting with one route is always the set of all routes. Using the encoding schema explained above, the action list enforcing the stable safe state is

$$(S20, =, F0); (S21, =, F0); (S22, =, F0) \qquad (1)$$

Let us now investigate how to encode the safety condition *"Whenever a route is in use by at least one tram, all conflicting routes must not be entered by trams at the same time."* We define separate transitions into stable safe state accompanied by the action list (1), one transition for every possible violation of the safety condition. To start with route 1, the in-use condition is expressed by the counter C20.1 associated with sensor G20.1 having a value > 0: The controller of route 1 will increment the counters C20.1, C20.2, C21.0 and C21.1 as soon as a tram passes the respective sensor. Identical values C20.1 = C21.1 indicate that the route is empty again, which leads to a reset of these counters to zero, performed by the route controller. The counters associated with sensors of the other routes are handled analogously. As a consequence, a violation of the above safety requirement involving route 6 is captured by condition list

$$(C20.1, >, 0) \; and \; (C24.1, >, 0) \qquad (2)$$

The route conflict table allows to derive the full set of these condition lists by generating one of the kind (2) for each pair of routes related by a $\circ$ in the table. The counters involved are the ones associated with the first sensor of each route, as defined in the route definition table. It turns out that six different condition lists can be derived. These can be reduced to three condition lists taking into account that and is symmetric. The two other condition lists are

$$(C20.1, >, 0) \; and \; (C22.1, >, 0) \qquad (3)$$
$$(C22.1, >, 0) \; and \; (C24.1, >, 0) \qquad (4)$$

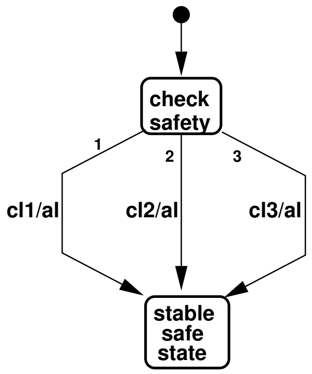

Fig. 4. The safety monitor state machine.

The resulting state machine for the safety monitor is shown in figure 4. In the figure al is the action list (1) and

$cl1$–$cl3$ are the condition lists (2)– (4), respectively. For the safety monitor, the priorities are assigned in an arbitrary way, since the different checks for conflicts are equally important.

V. Discussion

A. Conclusion

In this article, a domain-specific description formalism for railway and tramway control systems has been presented. A specification written in this formalism can be used to instantiate a generic framework, resulting in an executable software system based on interpreters operating on the configuration data compiled from the specification. Interpreters and configuration data can be represented in a formal language, thus associating a formal semantics to the domain specification.

B. Verification and Validation

Analyzing the specification process and the technique to obtain an executable software system, the following verification and validation objectives have to be considered:

1. Interlocking Table Validation. The construction of the interlocking tables introduced in Section III is intended as a manual design process based on the tramway network specification. The construction will be performed by an application expert with the knowledge about the routes to be "implemented" on the network. It has to be shown that no sensors or conflicts between routes have been overlooked and that the point positions required for each route have been recorded in a correct and complete way. Like in [7], we can use model checking techniques for this task.

2. Interpreter Verification. It has to be verified that the interpreters introduced in Section IV will never lead to an unsafe state in the railway network, provided that the interlocking tables have been defined correctly and have been properly compiled into the internal data representations. This verification task suggests formal mathematical proof techniques: Universally quantified over all admissible valuations of interlocking tables, the interpreters must ensure safety in the resulting control system. To discharge this verification objective, we are currently applying similar techniques to those described in [5], based on the RAISE tool and its integrated proof assistant. Since the formal intermediate language is so close to a programming language, we expect that the code verification against the formal interpreter specification will be straight forward or can even be omitted, since automatic integration testing with high coverage can be performed (see below).

3. Compiler Verification or Validation of Compilation Results. Provided that interlocking tables are complete and correct and that the interpreters have been implemented correctly, the next step is to justify that the compiler produces correct internal data representations from the interlocking tables. For this task, two alternatives are currently investigated: First, the compiler could be formally verified to be correct for all admissible table valuations, using similar techniques as for objective 2. Second, we could validate each specific compilation result by re-transformation into its original table representation. If the re-transformation is implemented with techniques that are sufficiently "orthogonal" to the compilation algorithms, it would be justified to consider the compilation as correct.

4. Automated Integration Testing. The final verification and validation step consists in testing the integrated hardware/software system: Our formal models do not consider the underlying hardware platform. As a consequence, it has to be investigated by testing methods whether the integration is consistent with the intended system behavior. Based on the interlocking table specifications, tests can be derived, executed and evaluated automatically, so that high coverage can be achieved on a low cost basis, see [8], [9], [10] for the basic testing approach currently applied to the control systems described in this article.

C. Related Work

The initial ideas of the work presented here have been published in [12]. They were stimulated by a collaboration between the authors and INSY GmbH (Berlin), a company specialized on the development of tramway control systems. Since then, the domain specification language, the implementation technique and the verification concept have been completed (the results are described in the present article), and feedback has been gained from a first prototype implementation.

The concepts described in this article are intended for control system designs, where the safety-critical control decisions are made in a centralized controller, while controllers located in trams are restricted to issuing route requests and displaying responses of the central controller to the engine driver. A distribution of the main control logic onto several controllers in a network is possible without design changes, if the underlying hardware and operating system provides a virtual shared memory abstraction, so that several controllers may operate on a global common view of the state space. This technique can be used if the networks connecting the controllers provide sufficient transmission speed. These speed requirements are usually not met by the mobile communication facilities used for data exchange between train controllers and stationary controllers, so train controllers cannot be equipped with a consistent view of the global state space. Therefore, if safety-critical control decisions should be performed in on-board train controllers, it is necessary to utilize another type of distributed control algorithm where each train only operates on the minimum amount of state information needed. A distributed algorithm of this type has been investigated and verified with respect to its safety properties by the authors in [5].

Our approach relies strongly on abstraction techniques. It should be noted that the ideas used in the *device abstraction* *layer* described in Section II are very similar to the techniques employed in general-purpose operating systems to facilitate the interaction of application programs with different hardware components by using device-independent application program interfaces (see, for example, [13]).

In the past, the idea to develop re-usable control algorithms which were "instantiated" by the configuration data of the concrete application has been applied quite often in an intuitive way. However, the systematic elaboration of this idea was mainly stimulated within the context of object-orientation and the definition and investigation of design patterns and frameworks [4]. The initial motivation for patterns and framework was to reduce development costs by means of systematic software re-use [6]. Within the formal methods communities, another advantage has been exploited: For many design patterns and frameworks, it is possible to elaborate *generic theories*, that is, collections of generic theorems which may be instantiated together with the pattern or framework, resulting in concrete theorems about correctness or safety properties of the concrete system. The publication [5] represents a generic theory for a distributed railway control framework; in [11], the application of generic theories for domain-independent design patterns have been described.

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 [3] as an example.

D. Acknowledgments

The specification, verification and validation aspects described in this article are investigated by the authors in 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* [1].

References

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

[2] H. Dierks. PLC-Automata: A New Class of Implementable Real-Time Automata. In M. Bertran and T. Rus (editors) *Transformation-Based Reactive Systems Development (ARTS'97)*, volume 1231 of Lecture Notes in Computer Science, pages 111-125. Springer-Verlag, 1997.

[3] W. J. Fokkink, G. P. Kolk and S. F. M. van Vlijmen. EURIS, a specification method for distributed interlockings. In *Proceedings of SAFECOMP '98*, volume 1516 of Lecture Notes in Computer Science, pages 296-305. Springer-Verlag, 1998.

[4] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[5] A. E. Haxthausen and J. Peleska. Formal Development and Verification of a Distributed Railway Control System. *IEEE Transactions on Software Engineering*, 26(8): 687-701, August 2000.

[6] I. Jacobsen, M. Griss and P. Jonsson. *Software Reuse.* ACM Press, 1997.

[7] M.P. Lindegaard, P. Viuf and A.E. Haxthausen. Modelling Railway Interlocking Systems. In E. Schnieder and U. Becker (editors): *Proceedings of the 9th IFAC Symposium on Control in Transportation Systems 2000, June 13-15*, pages 211-217, Braunschweig, Germany, 2000.

[8] J. Peleska. Test Automation for Safety-Critical Systems: Industrial Application and Future Developments. In M.-C. Gaudel and J. Woodcock (editors): *FME '96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of Lecture Notes in Computer Science, pages 39-59. Springer-Verlag, Berlin Heidelberg New York, 1996.

[9] J. Peleska. Formal Methods and the Development of Dependable Systems. Habilitationsschrift, Bericht Nr. 9612, Dezember 1996, Institut für Informatik und praktische Mathematik, Christian-Albrechts-Universität Kiel, 1997.

[10] J. Peleska and M. Siegel. Test Automation of Safety-Critical Reactive Systems. *South African Computer Jounal* 19:53-77, 1997.

[11] J. Peleska and Bettina Buth. Formal Methods for the International Space Station ISS. In E.-R. Olderog, B. Steffen (editors): *Correct System Design*, volume 1710 of Lecture Notes in Computer Science, pages 363-389, Springer-Verlag, 1999.

[12] J. Peleska, Alexander Baer and Anne E. Haxthausen. Towards Domain-Specific Formal Specification Languages for Railway Control Systems. In E. Schnieder and U. Becker (editors): *Proceedings of the 9th IFAC Symposium on Control in Transportation Systems 2000, June 13-15*, pages 147-152, Braunschweig, Germany, 2000.

[13] A. Rubini and J. Corbet. *LINUX Device Drivers.* O'Reilly, 2001.

[14] S. Schneider. An Operational Semantics for Timed CSP. *Information and Computation*, 116:193–213, 1995.

[15] Verified Systems International GmbH. *RT-Tester 4.1 – User Manual.* Bremen, 2001.