A formal approach for the construction and verification of railway control systems

Anne E. Haxthausen, Jan Peleska, Sebastian Kinder

Under consideration for publication in Formal Aspects of Computing

A Formal Approach for the Construction and Verification of Railway Control Systems

Anne E. Haxthausen[1], Jan Peleska[2] and Sebastian Kinder[3]

[1] Informatics and Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark
ah@imm.dtu.dk
[2] Department of Mathematics and Computer Science, Universität Bremen, Germany
jp@informatik.uni-bremen.de
[3] Department of Mathematics and Computer Science, Universität Bremen, Germany
kinder@informatik.uni-bremen.de

Abstract. This paper describes a complete model-based development and verification approach for railway control systems. For each control system to be generated, the user makes a description of the application-specific parameters in a domain-specific language. This description is automatically transformed into an executable control system model expressed in SystemC. This model is then compiled into object code. Verification is performed using three main methods applied to different levels. (0) The domain-specific description is validated wrt. internal consistency by static analysis. (1) The crucial safety properties are verified for the SystemC model by means of bounded model checking. (2) The object code is verified to be I/O behaviourally equivalent to the SystemC model from which it was compiled.

Keywords: domain engineering, domain-specific languages, code generation, formal methods, verification, railway control systems

1. Introduction

1.1. Motivation

The development of modern railway and tramway control systems represents a considerable challenge to both systems and software engineers: the goal to increase the traffic throughput while at the same time increasing the availability and reliability of railway operations leads to a demand for more elaborate safety mechanisms in order to keep the risk at the same low level that has been established for European railways until today.

Correspondence and offprint requests to: Anne E. Haxthausen, Informatics and Mathematical Modelling, Technical University of Denmark, bldg. 321, DK-2800 Lyngby, Denmark, e-mail: ah@imm.dtu.dk, fax: +45 45 930074, phone: +45 45 257510

The challenge is further increased by the demand for shorter time-to-market periods and higher competition among suppliers of the railway domain; both factors resulting in a demand for a higher degree of automation for the development, verification, validation and test phases of projects, without impairing the thoroughness of safety-related quality measures and certification activities. Motivated by these considerations, this paper describes an approach for the construction, verification and validation of railway control systems which has been elaborated by the authors and their collaborators during the last decade.

1.2. Two Problem Categories

A closer analysis shows that the problems to be solved can be structured according to two main categories.

(1) The design of novel *generic control algorithms* is stimulated by the availability of innovative technologies offering new possibilities for safe and reliable train control mechanisms. In this category the objective is to elaborate *generic theories*, that is, collections of theorems whose assumptions and implications are universally quantified over, say, railway networks of a certain type. As an example, we mention the investigation of distributed train control algorithms stimulated by the advent of mobile communication technologies, now making it possible to develop alternatives to the centralised interlocking paradigm. For the verification of these algorithms, mechanised (first-order or higher-order logic) proof support is desirable.

(2) The development and verification of *concrete system configurations* addresses a problem frequently arising in conventional developments: typically, railway control systems are nowadays constructed following the principles of object orientation and generics. The system is designed as a generic collection of classes, structured according to certain design patterns, collaborations and frameworks enforcing proven design principles and facilitating the utilisation of specific hardware technology. Concrete systems are instantiated from the generic collection using configuration data specifying the network to be controlled, available track elements (signals, sensors, points) etc. In spite of the elegance of this approach, it suffers from the flaw that – when conventionally tested for a limited number of different configurations – some software bugs are only revealed when new configuration variants are used. Additionally, the verification of configuration data requires a considerable effort, often necessitating customised verification tool sets which in turn have to be qualified. As a consequence, even minor configuration changes, induced, for example, by construction work on certain track sections, require complex verification processes. The solution to these problems would be an automated verification suite making it possible to *verify each concrete system instance together with its configuration*. Here, automation is a crucial requirement, since the conventional verification process currently only exercised once on generic system level would be far too time-consuming and expensive to be repeated on every concrete system instance. In this problem category verification does not involve universal quantification, since all configuration aspects are completely determined. As a consequence, model-based development combined with model or property checking, validated compilers or object code verifiers are the technical means of choice for the development and verification process.

In this article, we focus on the second problem category. For a detailed description of the first, the reader is referred to [HP00a] and further references listed there.

1.3. Domain-specific Approach

In recent years, domain-specific methods for software development have gained wide interest. One of the main objectives addressed by these techniques is the possibility for a given domain to reuse various assets when developing software, e.g. to develop a generic system from which one can instantiate concrete systems. Additionally, the use of domain-specific languages (DSLs) as front-ends for development tools is advocated. In contrast to general-purpose specification and programming languages, DSLs facilitate their utilisation by domain experts who are not specialists in the field of information technology, because they use the terminology of the application domain.

Inspired by these considerations, we have suggested an approach for efficient *construction* of a family of similar tramway or railway control systems in [HP02, HP03a] and exemplified it for a class of route-based tramway control systems. The idea is to provide a framework consisting of (1) a generic control system that can be instantiated with configuration data, (2) a DSL front-end for specifying application-specific parameters and (3) a generator from domain-specific descriptions into configuration data and instantiation rules for the concrete system. Hence, for each control system to be developed, application-specific parameters are described

in the domain-specific language and from this specification a control system can automatically be generated. An advantage of the front-end consists in the fact that it is much simpler to specify the parameters of a system in the domain-specific language and then apply the generator, than it is to program the configuration data directly. This speeds up the production time and reduces the risk of errors; furthermore, it can be done by domain experts without requiring the assistance of programming specialists.

While this approach clearly offers advantages, it requires careful work to *develop* such a language, generator and generic control system and to *automatically verify* that generated control systems are safe. For this purpose we use formal methods.

1.4. Automated Verification Approach

As "programming" language for the control systems we have chosen SystemC [GLMS02] which allows for formal reasoning based on an operational transition system semantics. SystemC serves both as a compilation target from semi-formal DSL descriptions to semantically well-founded formal specifications and as a high-level programming language which can be compiled into executable code. Our development approach prescribes that each time a SystemC control system model is generated and compiled into object code, verification shall be performed at two levels. (1) The SystemC control system model is verified to be safe by means of bounded model checking combined with an inductive proof strategy, and (2) the object code is verified to be a correct implementation of the SystemC control system model. For this purpose, the framework should provide support tools: a proof obligation generator and an object code verifier.

1.5. Development of Languages and Tools

For the development of a domain-specific language and support tools our suggestion is to follow the TripTych dogma by Dines Bjørner (see for instance [Bjø06c]) making a domain model describing the concepts of the application domain prior to the actual development of applications. Apart from separating the concern of describing *what there is* from the concern of describing *what there should be* (the applications), this ensures that different applications are based on the same conceptual understanding. Then from the domain-model one can establish a model of domain-specific descriptions and their static semantics, a model of the application generator and a model of the code verifier. For a case study we have formulated such models in the formal RAISE Specification Language, RSL, [RAI92].

1.6. Related Work

The work presented in this paper is based on results published in [HP00a, LVH00, PBH00, HP00b, HP02, HP03b, HP03a, GH03, HCD04, PGHD04, Ber06, PH07]. Our work has been inspired by Dines Bjørner's TripTych dogma and formal techniques for software development described in [Bjø06a, Bjø06b, Bjø06c, Bjø03a, Bjø06d, Bjø07].

The domain model used in our case study only includes concepts needed for our development framework. For domain models capturing a much broader collection of concepts for railways we consider Dines Bjørner's formal railway domain models (see e.g. [Bjø03c, BGH$^+$97]) as very promising candidates.

Object code verification has been investigated by several authors, see [PSS98] for an approach that has influenced our work in a considerable way. While our results have a similar formal basis – for example, our notion of I/O-equivalence is a specialisation of the "correct implementation relation" defined in [PSS98] – we exploit the specific restrictions of our model-based development framework in order to simplify the equivalence proofs in a considerable way.

For other complementary and competing approaches for the development and verification of railway control systems the reader is referred to the contributions in [TS03, ST04, ST07, EDD$^+$04], and for a survey of results and trends the reader is referred to the paper [Bjø03b].

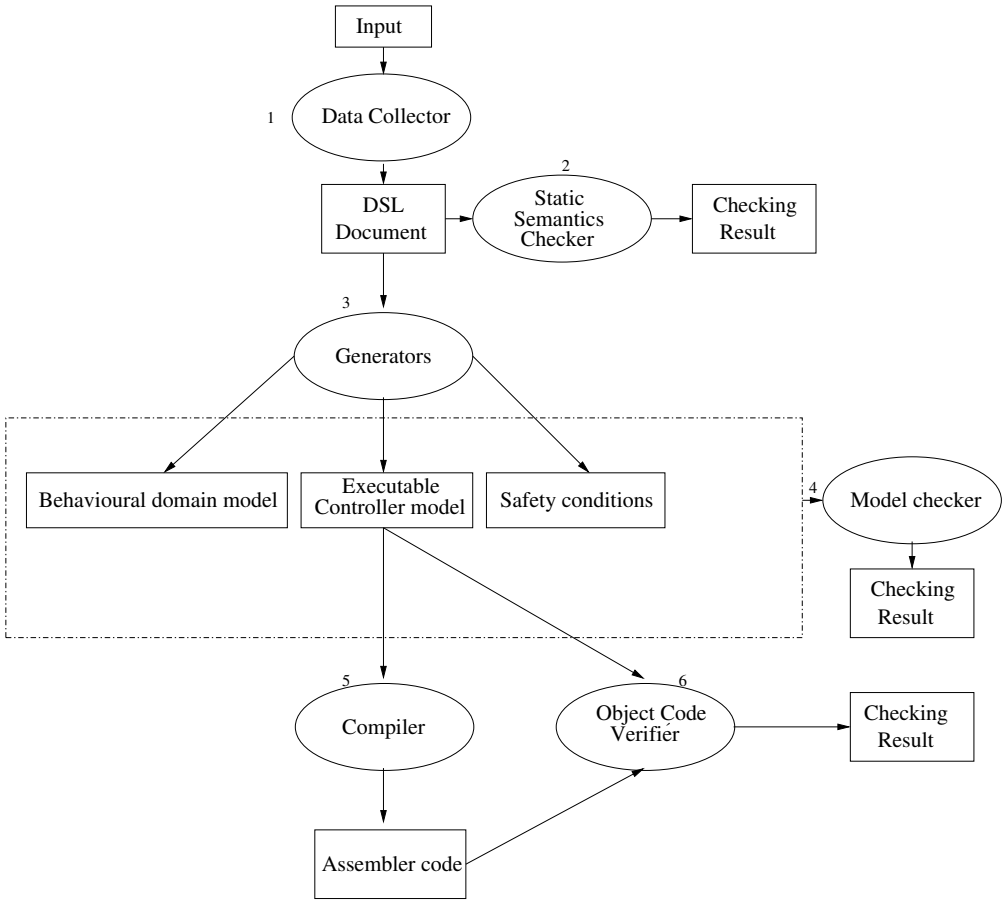

Fig. 1. Toolchain overview.

1.7. Paper Overview

First, in *Sect.* 2–3, we give an overview of our approach and informally describe a case study used to illustrate our approach. Then, in *Sect.* 4, we outline how a domain-specific description language for our case study can be formally developed from a static domain model. Next, in *Sect.* 5, we outline the development of application generators. In *Sect.* 6, we explain how the safety requirements can be verified. After that, in *Sect.* 7, we outline our approach for object code verification and we sketch how an associated code verifier can be formally developed. Finally, in *Sect.* 8, we discuss the work presented in this paper.

2. Method and Toolchain – Overview

Our approach requires a domain-specific language (DSL) and tools supporting the language. The main tool components required are:

1. A *data collector* for producing syntactically correct DSL text documents.
2. A *static semantics checker* for DSL documents.
3. *Generators* parsing DSL documents in order to create (1) executable controller models with transition system semantics (expressed in SystemC), (2) behavioural physical domain models with transition system semantics (SystemC), and (3) safety conditions (expressed in a subset of the temporal logic PSL [Acc04] that is a widely known industrial standard) for the concurrent composition of these two models.

4. A *bounded model checker* capable of verifying properties of composed controller and physical domain models written in SystemC.
5. A *compiler* for translating SystemC models into assembler code (since SystemC is embedded into C++, conventional compilers can be used for this task).
6. An *object code verifier* which, given a SystemC controller model and associated assembler code, verifies that the latter is a correct implementation of the former.

To create and verify a new control system users should apply these tools to go through the following steps illustrated in Fig. 1:

1. The railway specialists use the data collector to produce a syntactically well-formed DSL description $\mathcal{D}$ of domain-specific details of the system to be developed.
2. The static semantics checker checks that $\mathcal{D}$ is statically well-formed.
3. The generators automatically transform the domain-specific description $\mathcal{D}$ into a behavioural controller model $\mathcal{M}$, a behavioural physical domain model $\mathcal{P}$ (describing how uncontrolled physical devices are behaving) and a set of verification obligations Φ (safety properties as, for example, the requirement that trams should never meet within a track segment or on a point).
4. It is proved that the controller model $\mathcal{M}$ in concurrent combination with the physical model $\mathcal{P}$ satisfies the obligations Φ. This is done by means of an inductive proof strategy performed using bounded model checking techniques.
5. The controller model $\mathcal{M}$ is compiled into object code and data with conventional C/C++ compilers. This results in an assembler "model" $\mathcal{A}$.
6. The assembler "model" $\mathcal{A}$ is verified to be behaviourally equivalent to the controller model $\mathcal{M}$ using the object code verifier.
7. Finally the correctness of the hardware/software integration is automatically tested, following the concepts described in [BFPT06].

3. A Case Study

In a case study we have applied our approach to provide a framework for constructing and verifying a family of route-based tramway control systems. In particular we have designed a domain-specific language (DSL) and associated tools. Below, in Sect. 3.1, we informally explain the contents of descriptions ($\mathcal{D}$) in this domain-specific language and give a concrete example of such a description, and then, in Sect. 3.2, we outline the SystemC models ($\mathcal{M}$ and $\mathcal{P}$) and safety conditions (Φ) that the generators should produce from such descriptions. In Sect. 4 we explain how the language and associated static semantics checker were developed, and in Sect. 5 we explain the design of the primary generator: the application generator producing SystemC controller models. In Sect. 6 we describe our strategy for proving safety conditions Φ. We have successfully applied this strategy for the concrete sample DSL description from Sect. 3.1.

3.1. Domain-Specific Description

The basic requirements for avoiding tram collisions are that trams must only drive on predefined routes previously reserved and that two conflicting (overlapping) routes must not be reserved at the same time. As a consequence, controllers built to enforce these requirements depend on the railway network to be controlled and on a selection of predefined routes through that network. This implies that the associated domain-specific description should include network specifications and interlocking tables describing the routes.

Fig. 2 shows a DSL representation of a sample network, consisting of the following track components:

- *Sensors* detecting passing trams: G20.0, ..., G25.1
- *Controllable points*: W100, W102, W118.
- *Non controllable points*: shown by grey colour.
- *Signals*: S20, S21, S22.
- *Track segments*: (G20.0,G20.1), ..., (G25.0,G25.1), shown as solid lines between two sensors.

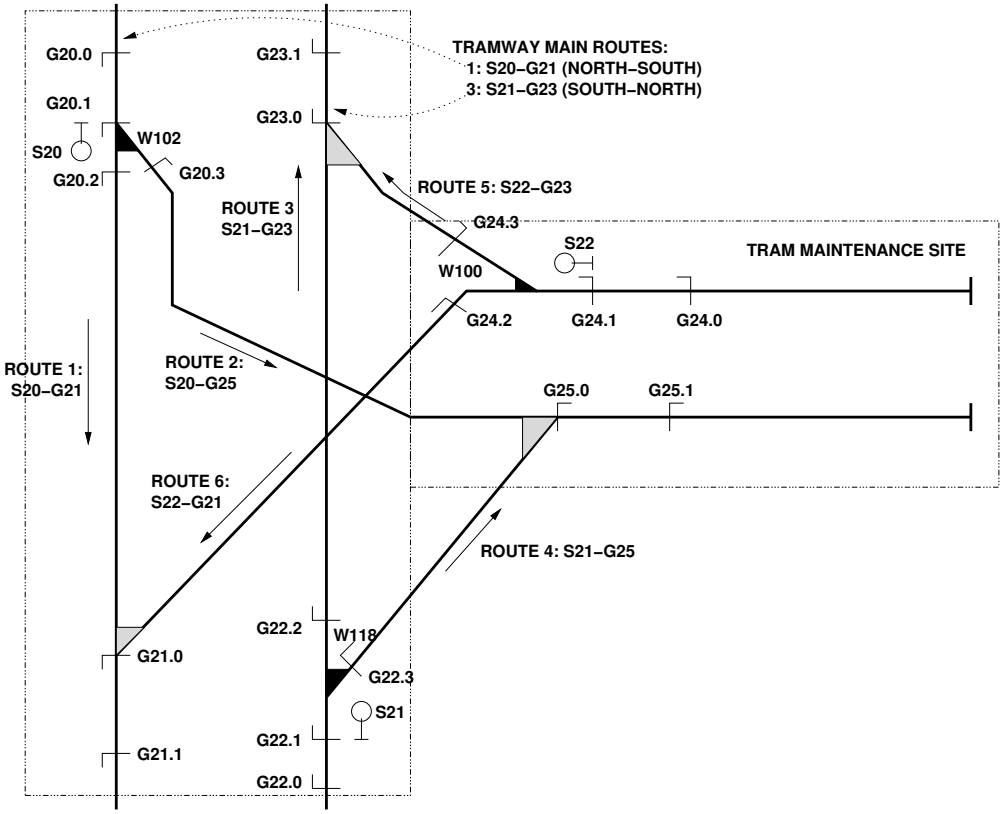

Fig. 2. The sample network.

The *interlocking tables*, which are also part of the DSL, comprise four items: (1) a *route definition table* specifying admissible routes through the network, (2) a *route conflict table* describing the routes not to be simultaneously allocated because they have a common route entry point (marked by "o" in the route conflict table) or overlap each other in another way (marked by "x"). (3) a *point position table* describing for each route how points should be set for its traversal, and (4) a *signal setting table* specifying for each route the name of its entry signal and the aspect it should be set to, in order to indicate that a tram is allowed to enter the route. In Fig. 3 the graphical representation of some sample interlocking tables for the network given in Fig. 2 are shown.

3.2. Generated SystemC Models and Safety Conditions

From DSL descriptions as that depicted in Fig. 2 and 3, the generators create SystemC models $\mathcal{M}$ and $\mathcal{P}$ together with the associated interface specifications and safety-related proof obligations Φ. For this purpose, the generators utilises a library of design patterns, so that architectural aspects, physical model, controller model and proof obligations are elaborated according to pre-defined schemes.

3.2.1. Interfaces

Interfaces are modelled according to the shared variable paradigm, to be realised using DMA or dual-ported RAM technology on all hardware interfaces (Fig. 4). Signal and point interfaces, for example, consist of three data fields: the requested state (controller → signal/point), the actual state (controller ← signal/point), and the time tick of the request so that a switching deadline can be checked in order to detect failed track elements. In SystemC the interfaces are modelled by the following variables:

- A clock `t` storing the current time, i.e. number of cycles performed by the controller.

Route Definition Table

Route Id	Sensor List
R1	<G20.1,G20.2,G21.0,G21.1 >
R2	<G20.1,G20.3,G25.0,G25.1 >
R3	<G22.1,G22.2,G23.0,G23.1 >
R4	<G22.1,G22.3,G25.0,G25.1 >
R5	<G24.1,G24.3,G23.0,G23.1 >
R6	<G24.1,G24.2,G21.0,G21.1 >

Route Conflict Table

Route Id	Conflicts with					
	R1	R2	R3	R4	R5	R6
R1	-	o	-	-	-	x
R2	o	-	x	x	-	x
R3	-	x	-	o	x	x
R4	-	x	o	-	-	-
R5	-	-	x	-	-	o
R6	x	x	x	-	o	-

Point Position Table

Route Id	W100	W102	W118
R1	-	STRAIGHT	-
R2	-	TURN	-
R3	-	-	STRAIGHT
R4	-	-	TURN
R5	TURN	-	-
R6	STRAIGHT	-	-

Signal Setting Table

Route Id	Signal Id	Signal Setting
R1	S20	GO
R2	S20	GO
R3	S21	GO
R4	S21	GO
R5	S22	GO
R6	S22	GO

Fig. 3. Interlocking tables.

- For each signal `S`: `reqsig[S]` storing *the requested state*, `actsig[S]` storing *the actual state*, and `reqsigtm[S]` storing *the time of request*.
- For each point `W`: `reqpt[W]` storing *the requested state*, `actpt[W]` storing *the actual state*, and `reqpttm[W]` storing *the time of request*.
- For each sensor `G`: `sen[G]` storing *the actual state* and `sentm[G]` storing *the time for the last LOW-to-HIGH transition*.
- `delta_s` storing a common switching deadline for signals.
- `delta_p` storing a common switching deadline for points.
- `delta_l` storing a common stabilisation deadline for sensors.

3.2.2. SystemC Model for Controller

The basic behavioural patterns of a control system generated for a network and collection of routes are as follows. When a tram approaches the network, a route is requested to be reserved. The control system makes a reservation for that route if no conflicting route has already been reserved. Then it allocates the route by requesting points to be switched into positions that allow traversal of the chosen route (as described by the point position table), and when the points have been switched it requests the entry signal to show a GO aspect (as described by the signal setting table) indicating that the tram may enter the route. As soon as the tram has passed the entry signal, the signal is requested to show STOP, and when the tram has left the route, the route is deallocated by removing its reservation.

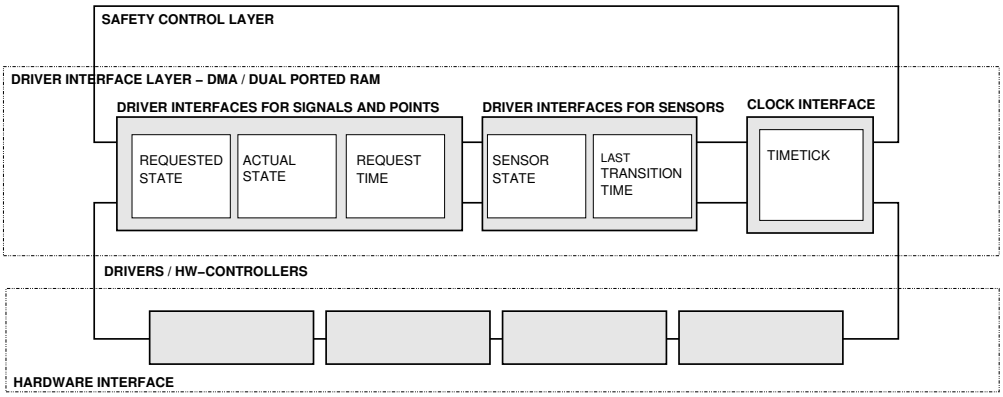

Fig. 4. Layered architecture and interfaces.

Each control system is implemented using a main loop, so that each execution cycle has four phases. In the *input phase* all current values of input interfaces (actual states) are copied to (global) shadow variables, in the *processing phase* interfaces are neither read nor updated, but global or local variables are processed. In the *wait phase* the system "spins" in an *active wait* loop without side effects (this is to ensure constant loop frequency), and in the *output phase* the states of global variables shadowing outputs are copied to the corresponding output interfaces (requested states). This usage of shadow variables is applied to establish a SystemC controller model which is close to its corresponding assembler implementation: in the assembler implementation, the input and output variables should not be changed before the end of an execution cycle, because changes in memory become immediately visible, while in SystemC these changes are first visible at the next execution cycle. In the remainder of this paper we will ignore the wait phase.

3.2.3. SystemC Model for Physical Domain

The physical model generated for a network consists of transition rules describing the behaviour of all sensors, signals and points of the network.

For each sensor `G` there is a virtual (i.e. not physically existing) counter `c[G]` storing the number of trams that have passed the sensor. Intuitively speaking, counters measure the "discrete flow" of trams through the network, while abstracting from concrete speed. In an empty network all counters are zero. A tram entering the network, say, at sensor G20.0, is modelled by the sensor state changing from `LOW` to `HIGH` and a counter state change from `c[G20.0] == 0` to `c[G20.0] == 1`. Flow propagation through the network is governed by a collection of sensor state transition rules: sensors G inside the network can change their state from `LOW` to `HIGH` and increment their counter values if the sum of all neighbouring sensors on routes directed to G is greater than G's current counter value. If, for example, G21.0 has counter value `c[G21.0] == 0`, is in state `LOW` and `c[G20.2] + c[G24.2] == 1` holds, this corresponds to the situation where a tram on route 1 or route 6 approaches, but has not yet reached, G21.0. The consecutive state change of G21.0 to `HIGH` is accompanied by a counter increment and reflects the situation where the tram has reached G21.0. Safety violations can be expressed by counters as well; `c[G21.0] == 0 && c[G20.2] == 1 && c[G24.2] == 1`, for example, would model the hazard where two trams simultaneously approach G21.0 from different routes. More details are presented in the safety requirements SF1 to SF5 below. More formally, sensor state changes from `LOW` to `HIGH` are modelled by rules following the pattern

```
if ( (sen[G] == LOW) && c[G] < <sum-incoming-counters> &&
     <signal-condition> && <nondeterministic-guard> ) {
    sen[G] = HIGH;
    sentm[G] = t;
    c[G] = c[G] + 1;
    <signal-action>;
}
```

In this pattern `<sum-incoming-counters>` stands for the counter sum of neighbouring sensors on routes

approaching G. The `<signal-condition>` applies for sensors guarded by signals: the sensor state may only change, that is, the tram may only pass, if the signal aspect is `GO`. This models the hypothesis that trams really stop at signals showing a `HALT` aspect, which is crucial for proving the desired safety properties. The `<nondeterministic-guard>` is an auxiliary input used to model all possible interleavings of tram movements: as long as this guard is false, the transition is not performed, modelling the situation where the tram has not yet reached the sensor. The time of the state change is stored in `sentm[G]` so that the latency of the sensor can be modelled: a state change from `HIGH` back to `LOW` may only occur after this latency interval has passed. Passing a sensor may be accompanied by a request for a signal state change; this is modelled in the `<signal-action>`.

To avoid overflow of the counters, there are additional rules for decrementing them after an occupying tram has completely passed through a route. For instance, for route 1 there is a rule stating that when `c[G21.1] == c[G20.2] > 0`, all the counters (`c[G20.0]`, `c[G20.1]`, `c[G20.2]`, `c[G21.0]`, `c[G21.1]`) of the route should be decremented by the value of the counter `c[G21.1]` (by which `c[G20.2]` and `c[G21.1]` become reset to 0). The condition `c[G20.2] > 0` expresses that there has been a tram on the route, and the condition `c[G21.1] == c[G20.2]` is intended to express that the route is now empty (i.e. the tram has left the route). This is true, as long as the safety invariant

```
(c[G20.2] == 0) || (c[G24.2] == 0)
```

holds. Similar "route-is-empty" conditions can be formulated for the other routes, and all of them require safety-related side conditions as the one above.

For each signal `S`, there is a transition rule having the following pattern:

```
if ( ((t >= reqsigtm[S] + delta_s) ||  <nondeterministic-guard>)
     && (reqsig[S] != actsig[S]) ) { actsig[S] = reqsig[S]; }
```

It states that within the specified switching deadline `delta_s` the actual state `actsig[S]` of the signal has to switch to the state `reqsig[S]` requested by the controller, if these states differ. To determine whether `delta_s` time units have elapsed, the time of the request `reqsigtm[S]` for this signal is compared to the time `t` of the tram control system. Again, `<nondeterministic-guard>` is a nondeterministic auxiliary input. Here it is set with an arbitrary value in each execution cycle and enables a state transition at arbitrary time ticks between the time of the request and the time limit for the transition. Here it is used to model situations where the signal switches to the requested state before the time bound for a correctly operating signal has been reached.

For points the transition rules are similar to the rules for signals.

3.2.4. Safety Conditions Φ

The major safety requirements Φ preventing collisions within the network boundaries marked by the entry signals are:

SF1 *Each segment not containing a point is occupied by at most one tram.* For the network shown in Fig. 2 there are 3 such segments: the 3 exit segments [G21.0, G21.1], [G23.0, G23.1], and [G25.0, G25.1].

SF2 *Each controllable point is occupied by at most one tram.* For the network shown in Fig. 2 there are 3 controllable points: W102, W100, W118.

SF3 *No two trams may approach the same sensor simultaneously from two directions.* For the network shown in Fig. 2 there are only three sensors that can be approached by trams from different directions: G21.0, G23.0, G25.0. (That it can't happen for the other sensors follows from the network configuration and the assumption that trams only enter the network at the signals and do not change direction.) Sensor G21.0, for example, can be approached by trams from G20.2 or G24.2, but not from G21.1.

SF4 *For two segments crossing each other at least one of them is empty.* For the network shown in Fig. 2 there are 3 crossings. E.g. there is a crossing between the segments [G20.3, G25.0] and [G22.2, G23.0].

SF5 *Each controllable point is empty when it is in a switching state.*

For a concrete network, these requirements can be formalised as conditions on the virtual counters `c[G]` introduced in the physical model $\mathcal{P}$ of the network. Then formula Φ can be specified as the conjunction of these conditions. We will now illustrate how some of these requirements have been formalised for the network

shown in Fig. 2. For instance, the **SF1** requirement that there is at most one tram on segment [G21.0, G21.1] can be expressed by the condition

```
c[G21.0] - c[G21.1] <= 1
```

as `c[G21.0]` indicates the number of trams that have entered the segment, and `c[G21.1]` indicates the number of trams that have left the segment. The **SF2** requirement that there is at most one tram on point W102 can be expressed by the condition

```
c[G20.1] - (c[G20.2] + c[G20.3]) <= 1
```

as `c[G20.1]` indicates the number of trams that have entered the point, and `c[G20.2] + c[G20.3]` indicates the number of trams that have left the point. The **SF3** requirement that at most one tram is approaching G21.0 from G20.2 or G24.2 can be expressed by the condition[1]

```
(c[G20.2] == 0) || (c[G24.2] == 0)
```

as `c[Gx.y] == 0` means that no tram has passed Gx.y after it was reset last time. The **SF4** requirement that one of the two crossing segments [G20.3, G25.0] and [G22.2, G23.0] is empty, can be expressed by the condition[2]

```
(c[G20.3]  == 0) ||  (c[G22.2] == 0)
```

This follows from the fact that trams only enter the segments [G20.3, G25.0] and [G22.2, G23.0] at G20.3 and G22.2, respectively, and therefore these segments are empty when the associated counters are zero. The **SF5** requirement that point W102 is empty when it is in a switching state can be expressed by the condition

```
(actpt[W102] !=  reqpt[W102]) -> (c[G20.1] - (c[G20.2] + c[G20.3])) == 0
```

For the network shown in Fig. 2 there are 15 such conditions in total, and Φ is the conjunction of these.

Observe that, in contrast to tramways, railways usually impose additional safety requirements concerning flank protection and shunting.

4. From Static Domain Model to Domain-specific Language

In this section we explain how a domain-specific description language for our case study can be formally developed from a static domain model using RSL. At the end of this development process, the RSL model obtained represents the *abstract syntax* and *static semantics* of the DSL under construction. It then only remains to associate the abstract syntactic elements with *concrete* syntax, in order to complete the DSL definition. The *behavioural semantics* is defined in a transformational way by means of the generator translating DSL specifications into SystemC models.

4.1. RSL Static Domain Model

We start by describing how a domain model can be established. The domain model covers the concepts of railway networks and routes. More general models would typically cover further concepts like time tables, but here we only present those concepts that are relevant for the development of the application considered in this paper. The model of each concept is *generic* (algebraic) in the sense that it defines *which* properties any concrete instance of the concept should have. The generic model can be instantiated to produce a *concrete* one defining *what* the specific properties are for that specific instance.

[1] This requirement could alternatively have been formalised by the condition `(c[G20.2] + c[G24.2]) - c[G21.0] <= 1`.

[2] Note that it would not have been correct to express the emptiness of e.g. [G20.3, G25.0] (on route 2) by the condition `c[G20.3] - c[G25.0] == 0` as G25.0 is also part of route 4, and therefore `c[G25.0]` can be incremented not only by trams going along route 2, but also by trams going along on route 4.

4.1.1. Generic Network Model

Any concrete network model should describe the topology of a railway network consisting of the physical components: segments, sensors, signals, and points.

In the generic model, for each kind of component, an abstract type of identifiers for its components is declared:

type Sensor, Point, Signal, Segment

Furthermore, signatures for functions that describe the relationship between the components are given. For instance, the following function gives the sensor at which a given signal is placed:

value sensor_of : Signal → Sensor

Finally, a number of axioms express requirements on these functions, i.e. impose restrictions on which network topologies are allowed. For instance, the following axiom requires that any two distinct signals are placed at distinct sensors:

∀ s1, s2 : Signal • s1 ≠ s2 ⇒ sensor_of(s1) ≠ sensor_of(s2)

To describe a *concrete network*, the elements of the types should be specified and the functions should be explicitly defined in such a way that the axioms are satisfied. In Sect. 4.2.2 we explain how the checking of axioms has been implemented in a tool.

4.1.2. Generic Route Model

An abstract type of identifiers for routes is declared:

type Route

We state the signature for a function that for a given route returns a list of those sensors which have to be passed in the stated order when travelling along the route:

value sensors_of : Route → Sensor*

A number of axioms express requirements to this function, i.e. impose restrictions on what is an allowed route. For instance, there must be a signal at the first sensor of any route:

∀ r : Route • ∃ s : Signal • sensor_of(s) = **hd** sensors_of(r)

4.2. Domain-specific Language

4.2.1. RSL Specification

The domain model is now extended with value declarations for each element to be part of a domain description. For each kind of physical component there is an element (all together providing a network description):

value
  sensors : Id-**set**,
  points : Id $\xrightarrow{m}$ (Sensor × Sensor × Sensor),
  signals : Id $\xrightarrow{m}$ Sensor,
  segments : Id $\xrightarrow{m}$ (Sensor × Sensor),
  crossings : (Segment × Segment)-**set**

The declarations give each element a name and a model-oriented type and hence provide an *abstract syntax* for the elements. As an example, the abstract syntax for the *sensors* element is $Id-$**set**. The *sensors* element contains the set of sensor identifiers, *points* maps each point identifier to the three sensors covering the point, *signals* maps each signal identifier to the sensor at which it is placed, *segments* maps each segment identifier

to the two sensors at the borders of the segment, and *crossings* contains the pairs of segments that cross each other. Similarly, for each kind of interlocking table there is an element[3]:

value
  rdt : Id $\overrightarrow{m}$ Sensor*,
  rct : Route → Route-**set** × Route-**set**,
  ppt : Route → Point $\overrightarrow{m}$ PointPosition,
  sst : Route → Signal × SignalSetting

In the route conflict table *rct*, for example, the pair $rct(r) = (c_1, c_2)$ identifies the routes with conflict types $\circ$ and $\mathtt{x}$, respectively (see Fig. 3): Route $r \circ r'$ if and only if $r' \in c_1$ and $r \ \mathtt{x} \ r''$ iff $r'' \in c_2$.

We chose identifiers to be texts:

type Id = **Text**

Now, the *Signal* type can be explicitly defined as containing the identifiers of the domain of the *signals* element:

type Signal = {| id : Id • id ∈ **dom** signals |}

The *Sensor*, *Point*, *Segment* and *Route* types can be defined in a similar way.

All functions from the domain model can now be explicitly defined in terms of the element values. In this way the axioms (that refer to these functions) from the domain model now impose well-formedness conditions on the elements of a language description. Additional axioms that impose well-formedness conditions on the *rct*, *ppt* and *sst* values are added, so that these axioms provide a *static semantics* for the DSL.

4.2.2. Concrete Syntax and Static Semantics Implementation

Two alternative solutions to the implementation of the concrete DSL have been made using the Extensible Markup Language *XML* [W3Cb] and the Unified Modelling Language *UML* [RJB04], respectively. Below we outline the XML solution that is documented in [DC04]. In [Ber06] it is described how the DSL is defined by a UML 2.0 profile in the second solution. The *concrete syntax* of the language has been defined by an XML document type definition (DTD). For the elements of the RSL abstract syntax, corresponding XML elements are defined. The *static semantics* has been implemented using the extensible style sheet language XSL [W3Cc]. In a systematic way each RSL axiom expressing a well-formedness requirement has been transformed into a template that tests whether the requirement is fulfilled. A GUI based *data collector* for creating DSL descriptions in the required XML syntax has been developed using XForms [W3Ca]. For the convenience of users, a *graphical representation* of DSL descriptions (XML documents) has been developed. This was done using XSLT and HTML. In Fig. 3 the graphical representation of some sample interlocking tables for the network given in Fig. 2 are shown.

5. Generating Applications from Domain-specific Descriptions

According to our method three generators taking a statically well-formed DSL description $\mathcal{D}$ as argument are required (Fig. 1). The primary one is the generator producing a control system model $\mathcal{M}$. The second generator produces the behavioural model $\mathcal{P}$ of the physical environment and the third one generates the safety properties Φ, to be checked to hold for the concurrent composition of the control system model $\mathcal{M}$ and the physical model $\mathcal{P}$. In this section we outline the basic concepts of the generator for $\mathcal{M}$. The other generators are designed in a similar way.

5.1. Components of the Controller Model Generator

The implemented generator for controller models consists of two parts:

[3] *rdt* for route definition table, *rct* for route conflict table, *ppt* for point position table, and *sst* for signal setting table.

1. A configurable library of generic code that is re-usable for all control systems to be generated: the code comprises generic versions of the control algorithms, the data structures carrying dynamic state information needed for performing control decisions, and the static configuration data structures.
2. A parser that takes a domain-specific description as input and returns concrete configuration data and instantiation parameters for the generic algorithms.

We have selected SystemC [GLMS02] as the target language for the generator, since it is associated with a formal transition system semantics and can be directly compiled into executable code [PGHD04, Ber06]. As a consequence, the original DSL descriptions "inherit" formal behavioural semantics from the transformations performed by the generator.

5.2. Generic Configurable Library

Since the SystemC code is automatically generated, the emphasis of the coding structure – whose layout is already fixed in the configurable library – lies on easy verifiability and efficient executability: the control structures of the generic algorithms utilise generic data structures (global arrays) and generic parameters. For example, if reservation of route i requires i to be requested, not already reserved, and excludes simultaneous reservation of conflicting routes j, then the reservation is performed by the following generic code structure:

```
bool mayReserve = requested[i] && ! reserved[i];
for (int j = 0; j < NUM_ROUTES; j++)
   mayReserve = mayReserve && ! (conflict[i][j] && reserved[j]) ;
if ( mayReserve )
   reserved[i] = 1;
```

where `requested` and `reserved` are arrays that keep parts of the dynamic control states, `conflict` is an array encoding the route conflict table and `NUM_ROUTES` is a constant specifying the number of routes. The actual value of `NUM_ROUTES` and contents of `conflict` differ from application to application and is defined by the concrete parameters and configuration data. This structure also ensures a close relationship between SystemC and assembler code which facilitates the object code verification in a considerable way.

The generic parameters referenced in the control algorithms of the library are of a very simple nature. They comprise number parameters, specifying the concrete quantities of sensors, signals, points and routes to be fixed for each system and offset parameters used for looking up specific routes and track elements in the static configuration data or in the dynamic control states.

The most important aspect of the static configuration data is the description of available track elements and route specifications. Routes are represented as sequences of index references to track elements, together with information about the required signal and point states to be enforced when allocating a route to a tram. This encodes the network description, route definition table, signal setting table and point position table of the domain-specific description. An additional array (`conflict`) is used for specifying the conflict relations between routes. This encodes the route conflict table of the domain-specific description.

5.3. DSL → SystemC Parser

The parser for producing concrete SystemC code from DSL descriptions proceeds in two passes. First, the number parameters are determined from the DSL and represented as C constant declarations. As a result the dimensions of all arrays used for storing static configuration data and dynamic state information are fixed. In the second pass the parser generates constant C array assignments carrying the configuration data and auxiliary offset information for looking up routes and track elements.

6. Verification of Safety Requirements

In this section we describe the verification step of our method. First we state the general verification objectives and assumptions, and then we describe the verification strategy. Finally we apply this for a concrete example.

6.1. Verification Objectives

When a controller model $\mathcal{M}$, a physical model $\mathcal{P}$, and safety conditions Φ have been generated from a DSL description $\mathcal{D}$, the next step according to our method is to verify that all possible $\mathcal{P}$ executions, when controlled by the model $\mathcal{M}$ executed in parallel, respect the safety conditions Φ at any time. This is written $(\mathcal{P} \parallel \mathcal{M})$ **sat** $G(\Phi)$. $\mathcal{M}$ and $\mathcal{P}$ are SystemC models as described in Sect. 3.2 and Φ is a PSL proposition over the state variables of $\mathcal{P}$ as described in Sect. 3.2.4. G is the PSL/LTL temporal "Globally" operator, so that $G(\Phi)$ means "Φ holds in all states reachable after an initialisation".

6.2. Verification Assumptions

Our verification strategy is driven by the following assumptions:

- It is assumed that the railway network description in $\mathcal{D}$ is complete and correct.
- No assumptions about the correctness of interlocking tables in $\mathcal{D}$ are made.
- The DSL description $\mathcal{D}$ inherits its formal behavioural semantics from the SystemC models which are automatically generated from it. As a consequence, no refinement proofs are required to ensure consistency between internal SystemC models and high-level DSL description.
- No assumptions about the correctness of the generators are made.
- The rules how trains can move in the uncontrolled network (physical model $\mathcal{P}$) are complete and correct. In particular, it is assumed that trains only enter the network at the entry signals, they stop at signals in HALT state and do not change direction.
- The generated safety conditions Φ are complete and correct.

Completeness of safety conditions Φ can be justified by means of a refinement proof starting with a high abstraction of safety conditions which explicitly relates trams to track segments, and is therefore easy to validate. The high abstraction is then refined to the expression of Φ in terms of sensor counter conditions.

6.3. Verification Strategy

Since we are not assuming that generators and interlocking tables are correct, an universal "once-and-for-all" verification is impossible: each system instance has to be verified with its concrete configuration data. As a consequence, it is desirable to elaborate a verification strategy which can be executed in an automated way.

With respect to full automation the *model checking* approach for $(\mathcal{P} \parallel \mathcal{M})$ **sat** $G(\Phi)$ seems attractive. It is well known, however, that conventional model checking would lead to state explosions for train control tasks of realistic size. As a consequence, we have adopted a *bounded model checking* strategy combined with inductive reasoning. To prove that Φ always holds we use the following inductive principle called *k-induction*:

1. First it is proved that $\Phi \wedge \Psi$ holds for the $k > 0$ first execution cycles after initialisation, i.e. $\Phi \wedge \Psi$ holds for $k > 0$ successive[4] states $\sigma_0, \ldots, \sigma_{k-1}$ of which σ_0 is the initial state of $(\mathcal{P} \parallel \mathcal{M})$.
2. Next the following is proved for an arbitrary execution sequence of $k+1$ successive states $\sigma_t, \ldots, \sigma_{t+k}$ of which the first σ_t is an arbitrary state (reachable or not from the initial state σ_0): if $\Phi \wedge \Psi$ holds in the k first states $\sigma_t, \ldots, \sigma_{t+k-1}$, then $\Phi \wedge \Psi$ will also hold for the $k+1^{st}$ state σ_{t+k}.

Here Ψ is an auxiliary property that holds for reachable states. (Note that Ψ is simultaneously proved by the given induction principle.) The proofs of the base case and the induction step are performed by a bounded model checker tool described in [DG05]. This tool treats the two proof obligations by exploring corresponding propositional satisfiable problems and solving these by a SAT solver. Note that the induction steps argue over an execution sequence of k+1 states of which the first state, σ_t, may be unreachable, although it would have been sufficient for the truth of $G(\Phi)$ only to consider sequences for which σ_t is reachable. For sequences starting at an unreachable state, the induction step may fail and the property checker give a false negative. To avoid this the desired property Φ is strengthened with an auxiliary property Ψ that is false for those unreachable states, σ_t, for which the induction step would otherwise fail.

[4] Two states σ_i and σ_{i+1} are successive, if there is a transition from σ_i to σ_{i+1} according to $(\mathcal{P} \parallel \mathcal{M})$.

6.3.1. Auxiliary Condition Ψ

The auxiliary condition Ψ is a conjunction of state relations. Below we give examples of some of these relations. As mentioned earlier, these are needed as assumptions in the induction step of the proof of $G(\Phi)$, in order to rule out unreachable states that would have given rise to false negatives otherwise.

Example 1. Since the model is implemented as a timed state transition system, time consistency has to be established as a part of Ψ. Time consistency means that the current time `t` kept by the system is always larger than or equal to the time stamps in the interfaces `reqsigtm[S]`, `reqpttm[W]`, and `sentm[G]` of each signal `S`, point `W`, and sensor `G`, respectively. For example, for signal `S20` of the network in Fig. 2 this means:

```
reqsigtm[S20] <= t
```

Example 2. For each sensor `G`, the controller model has a counter `cc[G]` that plays the same role for the controller as the virtual counter `c[G]` does for the physical model. Both counters are initially 0. At the same time as the sensor becomes high (i.e. `sen[G] = HIGH`), the virtual counter `c[G]` is incremented by 1. The controller will first detect that the sensor is high one execution cycle later and increment its counter `cc[G]` by 1 in that cycle. Hence, either the two counters, `c[G]` and `cc[G]`, have the same value, or `c[G] == (cc[G] + 1)` in which case we are just one time unit after the sensor became high (i.e. `t == sentm[G] + 1`). The following relation expresses this and is included in Ψ:

```
((c[G] == cc[G]) || ((t == sentm[G] + 1) && (c[G] == cc[G] + 1)))
```

6.3.2. Additional Assumptions

To avoid overflow of the clock `t` in the proof of the induction step, we additionally assume `t < t_max` in the first state, σ_t, where `t_max` is chosen such that `t_max + k` is less than the maximal integer in the integer type of `t`. At the same time `t_max` is chosen such that it is larger than the maximal number of cycles that a controller can perform during a day from when it is started until it is stopped.

Note that the time consistency relations defined in Example 1 together with the assumption `t < t_max` also ensure that no overflow will occur in the `reqsigtm[S]`, `reqpttm[W]`, and `sentm[G]` interfaces.

6.4. Application Example

For the controller that can be generated from the sample network and interlocking tables in Sect. 3, we have used the strategy presented above to prove that it is safe. A value of $k = 3$ sufficed to carry out the induction. With `t_max = 4 billion`, it took the model checker 391.53 seconds to do the proof.

7. Object Code Verification

In this section we outline our approach for object code verification that is described in detail in [PH07], and we sketch how an associated code verifier can be formally developed.

7.1. Motivation

Automated object code verification for safety-critical control systems is motivated by the fact that applicable standards for these safety-critical applications, e.g. for railways [ECfES01], require a substantial justification with respect to the consistency between high-level software code and the object code generated by the applied compilers.

7.2. Approach

The conventional approach for this is *compiler validation*: "once-and-for-all" it is validated that the compiler for any input produces object code that is a correct implementation of that input. However, such an approach

is very time-consuming, especially if it should be done formally (see e.g. [GZ99] for techniques for that), and furthermore it has to be performed again whenever modifications of the compiler have been performed. An alternative to compiler validation is *object code verification*: each time object code is generated (by an arbitrary compiler), the generated object code is verified to be a correct implementation of the high-level software code. Object code verification has the advantage that it is independent of changes in the compiler and it can be fully automated and reasonably fast, if the compiled code originates from high-level programs strictly adhering to certain programming patterns as is the case for our generated SystemC models.

Our specific approach to object code verification is as follows: to prove that an assembler program (object code) $\mathcal{A}$ is a correct implementation of the SystemC controller model $\mathcal{M}$ from which it is generated, one should map (see Sect. 7.5) $\mathcal{A}$ and $\mathcal{M}$ to their behavioural models $\mathcal{T}(\mathcal{A})$ and $\mathcal{T}(\mathcal{M})$ given in terms of some common semantic foundations (I/O-Safe Transition Systems to be explained in Sect. 7.3) and then prove that $\mathcal{T}(\mathcal{A})$ and $\mathcal{T}(\mathcal{M})$ are I/O equivalent (modulo a variable renaming defined in Sect. 7.6) by applying transformations that have been proved "once-and-for-all" to preserve I/O behaviour (see Sect. 7.4).

7.3. Common Semantic Foundations: I/O-safe Transitions Systems

In this section we introduce our notion of *I/O-safe transitions systems (IOTS)* and our notion of *I/O equivalence* between IOTS.

7.3.1. Abstract Syntax and Static Semantics of IOTS

I/O-safe transitions systems (IOTS) are closely related to transition diagrams (as defined e.g. in [MP92]) consisting of (1) a set of variables for which initial values are given by an initial state, (2) a set of locations one of which is designated as the initial location l_0, and, (3) a set of transition rules. Variables are classified into input, output and processing variables. A transition rule from one location l_1 to another location l_2 is specified by a guard that is a quantifier-free predicate over the variables and by a multiple assignment $(v_1, \ldots, v_n) := (e_1, \ldots, e_n)$, where $v_1, \ldots, v_n$ are variables and $e_1, \ldots, e_n$ are expressions over the given set of variables. A new characteristic of IOTS consists in the fact that locations can be partitioned into pairwise disjoint sets of input, output and processing locations, and we put further constraints on the allowed use of variables in guards and expressions in an IOTS: guards must only use processing variables, for transitions into input locations the assignments must only read input variables and make assignment to processing variables, for transitions into processing locations the assignments must only read processing variables and make assignment to processing variables, and, for transitions into output locations the assignments must only read processing variables and make assignment to output variables.

One can obviously specify an abstract syntax of IOTS in RSL :

```
type
  IOTS ::
    vars : Var-set
    initstate : State
    locs : Loc-set
    initloc : Loc
    trans : TransitionRel-set,
  TransitionRel = Loc × Guard × Assign × Loc,
  Assign :: al : (Var × Expr)*,
  Expr == mk_Const(i : Int) | mk_Var(v : Var) | mk_Sum(e1 : Expr, e2 : Expr) | ...,
  Guard == TRUE | ...,
```

where *Guard* and *Expr* are the abstract syntax of guards and expressions, respectively, for space reasons not completely specified here. For variables and locations two abstract types are used, each having an observer function *mode* that returns the mode (input, output or processing) of variables and locations, respectively:

```
type Var, Loc
value mode : Var → Mode, mode : Loc → Mode
type Mode == IN | OUT | PROC
```

We also introduce a well-formedness predicate for IOTS formalising all the conditions on the use of variables and locations stated informally above:

```
value
  is_wff : IOTS → Bool
  is_wff(iots) ≡ dom initstate(iots) = vars(iots) ∧ initloc(iots) ∈ locs(iots) ∧ ...
```

For instance, the predicate checks that the initial state of an IOTS gives initial values to the variables in its variable set and that the initial location is in its location set.

7.3.2. Dynamic Semantics of IOTS

Now we define a dynamic semantics of IOTS. It involves states. A state σ for an IOTS is a valuation of its variables:

```
type State = Var →m Int
```

Each transition relation specification of an IOTS induces a state transformer:

```
value
  eval : TransitionRel → (State → State)
  eval(l, g, a, l′)(σ) ≡ if eval(g)(σ) then eval(a)(σ) else σ end
```

Here $eval(g)(\sigma)$ and $eval(a)(\sigma)$ are the standard extensions of the valuation σ to guards g and assignments a, respectively. It should be noted that we have defined the evaluation, $eval(mk\_Var(v))(\sigma)$, of variables $v \notin \mathbf{dom}\ \sigma$ to give the default value 0 to avoid partial evaluation functions. For well-formed IOTS this does no harm as only variables in $v \in \mathbf{dom}\ \sigma$ are evaluated by the semantics defined below.

The *semantics* of an IOTS is the set of its possible runs. A possible *run* of an IOTS is a non-empty sequence of pairs of locations and states such that the first location is its initial location, the first state is its initial state, and that for each consecutive pairs in the list there is a transition relation in the IOTS from the location of the first pair to the location of the second pair so that the associated state transformer maps the non input part of the state of the first pair to the non input part of state of the second pair:

```
type Run = (Loc × State)*

value
  eval : IOTS → Run-infset
  eval(iots) ≡
    { r | r : Run •
        len r > 0 ∧
        let (l0, σ_0) = hd r in
          l0 = initloc(iots) ∧ σ_0 = initstate(iots)
        end ∧
        (∀ i : Int • i > 0 ∧ i < len r ⇒
          let
            (l_i, σ_i) = r(i), (l_i′, σ_i′) = r(i+1)
          in
            dom σ_i′ = vars(iots) ∧
            (∃ (l, g, a, l′) : TransitionRel •
             (l, g, a, l′) ∈ trans(iots) ∧
             l = l_i ∧ l′ = l_i′ ∧
             eval(l, g, a, l′)(σ_i)\ivars(σ_i) = σ_i′\ivars(σ_i))
          end
        )
    },

  ivars : State → Var-set
  ivars(σ) ≡ {v | v : Var • v ∈ dom σ ∧ mode(v) = IN}
```

7.3.3. IOTS Equivalence

We are now going to define a notion of I/O equivalence of IOTS. In order to do that, we first need to define some auxiliary notions. An *I/O restriction* of a run is the restriction of the run to pairs where the location is an input or output location, and for these pairs the states are restricted to input and output variables only:

```
IOrestrict : Run → Run
IOrestrict(r) ≡ ⟨ (l, σ / { v | v : Var • mode(v) ∈ {IN,OUT} }) | (l,σ) in r • mode(l) ∈ {IN,OUT} ⟩
```

An *I/O restriction* of a set of runs rs is the set of I/O restrictions of the runs in the set rs:

```
IOrestrict : Run-infset → Run-infset
IOrestrict(rs) ≡ {IOrestrict(r) | r : Run • r ∈ rs}
```

An *I/O map* ρ is a bijective, mode preserving variable mapping between I/O variables:

```
type IOMap = {| ρ : Var ~m~> Var • bijective(ρ) ∧ IOmodepreserving(ρ) |}
value
  bijective : (Var ~m~> Var) → Bool
  bijective(ρ) ≡ (∀ v2 : Var • v2 ∈ rng ρ ⇒ (∃! v1 : Var • v1 ∈ dom ρ ∧ ρ(v1) = v2)),

  IOmodepreserving : (Var ~m~> Var) → Bool
  IOmodepreserving(ρ) ≡
    (∀ v : Var • v ∈ dom ρ ⇒
       mode(ρ(v)) = mode(v) ∧ mode(v) ∈ {IN,OUT})
```

Two I/O restricted runs are *I/O equivalent* wrt. an I/O map if they have (1) the same length, (2) the same order of input locations and output locations, and (3) their states agree on input variables and output variables modulo the I/O map:

```
equiv : Run × Run × IOMap → Bool
equiv(r1_io, r2_io, ρ) ≡
  len r1_io = len r2_io ∧
  (∀ j : Int • j > 0 ∧ j ≤ len r1_io ⇒
     let (l1, σ1) = r1_io(j), (l2, σ2) = r2_io(j) in
       mode(l1) = mode(l2) ∧
       σ1 = σ2 ° ρ
     end
  )
```

Finally, we can define two IOTS to be *I/O equivalent* wrt. an I/O map ρ, if there is a bijection γ between I/O equivalent I/O restrictions of runs of the two IOTS:

```
equiv : IOTS × IOTS × IOMap ~→ Bool
equiv(iots1, iots2, ρ) ≡
  (∃ γ : Run ~m~> Run •
    dom γ = IOrestrict(eval(iots1)) ∧ rng γ = IOrestrict(eval(iots2)) ∧
    (∀ r1, r2 : Run • {r1, r2} ⊆ dom γ ∧ r1 ≠ r2 ⇒ γ(r1) ≠ γ(r2)) ∧
    (∀ r : Run • r ∈ dom γ ⇒ equiv(r, γ(r), ρ))
  )
pre dom ρ = iovars(iots1) ∧ rng ρ = iovars(iots2),

iovars : IOTS → Var-set
iovars(iots) ≡ {v | v : Var • v ∈ vars(iots) ∧ mode(v) ∈ {IN,OUT}}
```

For the identity variable mappings id we just write $equiv(iots1, iots2)$ rather than $equiv(iots1, iots2, id)$.

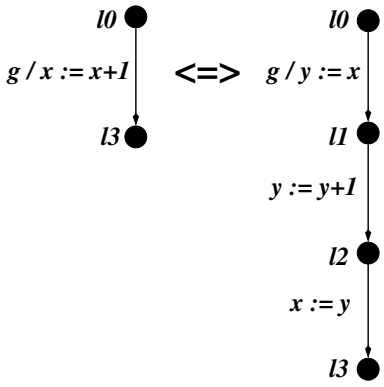

Fig. 5. A transformation rule.

7.4. IOTS Transformation Rules

We have developed a collection (see [PH07]) of transformation rules between IOTS patterns and proved by hand that any instance of the rules gives rise to a transformation that preserves I/O behaviour. The RSL formulation of the IOTS concepts in the previous section now enables us to formalise these proofs. As an example, there is a rule stating that an IOTS *iots*1 can be transformed into an equivalent IOTS *iots*2 by replacing the transition shown on the left hand side of Fig. 5 with the three transitions shown on the right hand side of Fig. 5, or vice versa, provided that (1) $l1$ and $l2$ are not locations of *iots*1, x and y are local variables, and (2) in any path emanating from location $l3$, the variable y is assigned before read. The rule is generic in locations $l0$, $l1$, $l2$, $l3$, variables x and y, and guard g. Proving this rule correct amounts to proving:

```
∀ iots1, iots2 : IOTS, l0, l1, l2, l3 : Loc, g : Guard, x, y : Var •
  {x, y} ⊆ vars(iots1) ∧ mode(x) = PROC ∧ mode(y) = PROC ∧
  {l0, l3} ⊆ locs(iots1) ∧ l1 ∉ locs(iots1) ∧ l2 ∉ locs(iots1) ∧
  (l0, g, mk_Assign(⟨(x, mk_Sum(mk_Var(x), mk_Const(1)))⟩), l3)
    ∈ trans(iots1) ∧
  vars(iots2) = vars(iots1) ∧
  initstate(iots2) = initstate(iots1) ∧
  locs(iots2) = locs(iots1) ∪ {l1, l2} ∧
  initloc(iots2) = initloc(iots1) ∧
  trans(iots2) =
    trans(iots1) \
      {(l0, g, mk_Assign(⟨(x, mk_Sum(mk_Var(x), mk_Const(1)))⟩), l3)}
    ∪
    {(l0, g,  mk_Assign(⟨(y, mk_Var(x))⟩), l1),
     (l1, TRUE,  mk_Assign(⟨(y, mk_Sum(mk_Var(y), mk_Const(1)))⟩), l2),
     (l2, TRUE,  mk_Assign(⟨(x, mk_Var(y))⟩), l3)
    } ∧
  assigned_before_read(iots1, l3, y)

  ⇒

  equiv(iots1, iots2)
```

Another example of a transformation rule is one stating that an IOTS *iots*1 can be transformed into an equivalent IOTS *iots*2 by replacing the transition shown on the left hand side of Fig. 6 with the four transitions shown on the right hand side of Fig. 6, or vice versa, provided that (1) $l1$, $l2$ and $l3$ are not locations of *iots*1, *aux*1 and *aux*2 are local variables, and (2) in any path emanating from location $l4$, the

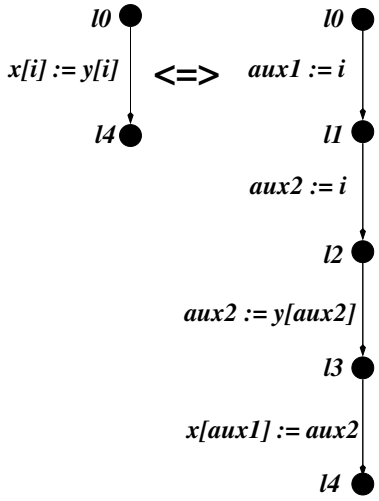

Fig. 6. Another transformation rule.

variables $aux1$ and $aux2$ are assigned before read. The rule is generic in locations $l0$, $l1$, $l2$, $l3$ and $l4$, and variables $aux1$, $aux2$, x and y.

7.5. IOTS Semantics of the Source and Target Languages

Any SystemC model $\mathcal{M}$ generated from a domain-specific description adheres, as explained in Sect. 3.2.2, to a set of simple programming patterns: it has a main loop consisting of phases each adhering to specific, restricted rules for allowed variable access, and furthermore each phase only consists of assignments, conditionals, and for statements. SystemC models $\mathcal{M}$ adhering to these programming patterns can therefore quite easily be given a behavioural semantics in terms of an IOTS. This semantics is far more simple than the complex semantic [MRR03] of general SystemC models.

Likewise, any assembler program $\mathcal{A}$ generated by applying a conventional C/C++ compiler to a SystemC model $\mathcal{M}$ (that has been generated from a domain-specific description $\mathcal{D}$) adheres to a set of simple "low-level patterns"[5] that makes it possible to give it an IOTS semantics. This follows from the fact that the controller model $\mathcal{M}$ adheres to the coding patterns explained above and the fact that compiler optimisations are not used in safety-critical applications.

In [PH07] IOTS semantics for generated SystemC models and IOTS assembler code was informally described. These semantics can be formalised in RSL by defining abstract syntax *Ccode* and *AssemblerCode* for generated SystemC models and assembler programs, respectively:

type Ccode = ...
type AssemblerCode = ...

and then defining semantic evaluation functions having the following signatures:

value $\mathcal{T}$: Ccode → IOTS
value $\mathcal{T}$: AssemblerCode → IOTS

We will not present a full RSL formalisation, but below we will describe the IOTS (RSL values) that each of the two evaluation functions return when applied to a given SystemC model and a given assembler program, respectively.

[5] For instance, the $\mathcal{A}$ program consists of a control loop that corresponds to the main loop of the SystemC program $\mathcal{M}$.

7.5.1. Semantics of SystemC Models

The behavioural semantics of a generated SystemC model $\mathcal{M}$ is an IOTS: $\mathcal{T}(\mathcal{M}) = (V^{\mathcal{M}}, \sigma_0^{\mathcal{M}}, L^{\mathcal{M}}, T^{\mathcal{M}})$ where the values $V^{\mathcal{M}}$, $\sigma_0^{\mathcal{M}}$, $L^{\mathcal{M}}$, and $T^{\mathcal{M}}$ are as described below.

The set of variables $V^{\mathcal{M}} = V_I^{\mathcal{M}} \cup V_O^{\mathcal{M}} \cup V_P^{\mathcal{M}}$ where

- $V_I^{\mathcal{M}}$ is the set of input variables. It consists of the interfaces (introduced in Sect. 3.2.1) that the controller model $\mathcal{M}$ uses to read the state of hardware devices, e.g. `actpt[n]` (actual state of point number `n`).
- $V_O^{\mathcal{M}}$ is the set of output variables. It consists of the interfaces (introduced in Sect. 3.2.1) that the controller model $\mathcal{M}$ uses to send requests to hardware devices, e.g. `reqpt[n]` (for sending requested state to point number `n`).
- $V_P^{\mathcal{M}} = V_G^{\mathcal{M}} \cup V_L^{\mathcal{M}}$ is the set of processing variables.
- $V_G^{\mathcal{M}}$ is the set of global variables and constants declared in $\mathcal{M}$. It consists of a shadow variable (see Sect. 3.2.2) for each input and output variable, e.g. `actptNext[n]` and `reqptNext[n]`, internal state variables such as the array entry `reserved[n]` (that keeps track of the reservation status of route number n), and constants like conflict table entries `conflict[i][j]` used internally in the processing phase.
- $V_L^{\mathcal{M}}$ is the set of local variables `i, j, r, ...` in $\mathcal{M}$ used as loop counters and temporary variables.

The initial state $\sigma_0^{\mathcal{M}}$ is derived in the obvious way from the initialisation part of the SystemC model $\mathcal{M}$. For local variables $x \in V_L^{\mathcal{M}}$ the initial valuation is undefined, but it is made sure by means of static analysis that no local variable is read before having been written to.

The set of locations $L^{\mathcal{M}}$ consists of all labels that are associated with the statements in $\mathcal{M}$ when using the following labelling procedure. Each statement s in $\mathcal{M}$ is given two labels: a pre-label l_s and a post-label l'_s. Furthermore, each for statement is given a third label l''_s. In general the labels associated with the statements must be distinct, however there are some exceptions. For instance, for any compound statement $\{\ s_1 \dots s_i\ \ s_{i+1} \dots s_n\}$ its pre-label l must be equal to the pre-label of its first statement (i.e. $l = l_{s_1}$), its post-label l' must be equal to the post-label of its last statement (i.e. $l' = l_{s_n}$), and for any two consecutive statements s_i and s_{i+1} the post-label of s_i must be identical to the pre-label of s_{i+1} (i.e. $l'_{s_i} = l_{s_{i+1}}$). Another exception is the requirement that the post-label l' of any conditional `if(b) s` must be equal to the post-label of `s` (i.e. $l' = l'_s$).

For each location $l \in L^{\mathcal{M}}$ its mode can be derived by investigating the mode of variables appearing in those transitions (in the set $T^{\mathcal{M}}$ described below) that go into l (i.e. are of the form $(..., ..., ..., l)$). The set of transitions $T^{\mathcal{M}}$ is constructed by deriving one or more transitions from each statement in $\mathcal{M}$:[6]

- A compound statement leads to the union of the transitions that its constituent statements lead to.
- An assignment `x = e;` with pre-label l and post-label l' leads to a transition $(l, \texttt{true}, x := e, l')$.
- A guarded statement `if(b) s` with pre-label l and post-label $l' = l'_s$ leads to the transitions associated with `s` as well as the following transitions: (l, b, ε, l_s) and $(l, \neg b, \varepsilon, l')$, where ε denotes the empty assignment which does not change any variable valuation.
- A for statement `for (i=0; i < c ;i++) s` with pre-label l, post-label l', and third label l'' leads to the transitions associated with `s` as well as the following transitions: $(l, \texttt{true}, i := 0, l'')$, $(l'', i >= c, \varepsilon, l')$, $(l'', i < c, \varepsilon, l_s)$, and $(l'_s, \texttt{true}, i := i + 1, l'')$.

In this way the input phase of $\mathcal{M}$ leads to transitions into input and processing locations, the processing phase leads to transitions into processing locations, and the output phase leads to transitions into output and processing locations.

7.5.2. Semantics of Assembler Code

The behavioural semantics of an assembler program $\mathcal{A} = g(\mathcal{M})$, where $\mathcal{M}$ is a generated SystemC model, is an IOTS: $\mathcal{T}(\mathcal{A}) = (V^{\mathcal{A}}, \sigma_0^{\mathcal{A}}, L^{\mathcal{A}}, T^{\mathcal{A}})$, where the values $V^{\mathcal{A}}$, $\sigma_0^{\mathcal{A}}$, $L^{\mathcal{A}}$, and $T^{\mathcal{A}}$ are as described below.

The set of variables $V^{\mathcal{A}} = V_I^{\mathcal{A}} \cup V_O^{\mathcal{A}} \cup V_P^{\mathcal{A}}$ where

[6] To make the description of this derivation easier to read, we use concrete syntax for SystemC statements, IOTS guards, and IOTS assignments.

- $V_I^{\mathcal{A}} = \{\texttt{x(,n,4)} \mid \texttt{x[n]} \in V_I^{\mathcal{M}}\}$ is the set of input variables.
- $V_O^{\mathcal{A}} = \{\texttt{x(,n,4)} \mid \texttt{x[n]} \in V_O^{\mathcal{M}}\}$ is the set of output variables.
- $V_P^{\mathcal{A}} = V_G^{\mathcal{A}} \cup V_L^{\mathcal{A}}$ is the set of processing variables.
- $V_G^{\mathcal{A}} = \{\texttt{x(,n,4)} \mid \texttt{x[n]} \in V_G^{\mathcal{M}}\}$.
- $V_L^{\mathcal{A}} = V_L^{\mathcal{M}} \cup REGS \cup FLAGS \cup SADDR$.
- $REGS$ contains all symbols `%eax, %edx, ...` denoting registers.
- $FLAGS$ contains the symbols `ZF, SF, PF, ...` for zero flag, sign flag, parity flag and others.
- $SADDR$ contains stack address symbols used for auxiliary variables.

It is easy to see that there is a 1-1 relationship between the variable symbols in $V^{\mathcal{A}}$ and $V^{\mathcal{M}}$, except that $V_L^{\mathcal{A}}$ contains additional assembler-specific variable symbols. All variables in $V^{\mathcal{A}}$ except the local variables are represented as arrays. Expression `actpt(,n,4)` (where `n` is a constant), for example, denotes the contents of the 4-bytes memory cell at memory byte address $\texttt{actpt} + 4 \cdot n$.

The set of locations $L^{\mathcal{A}}$ consists of all labels that are associated with the instructions in $\mathcal{A}$ when using the following labelling procedure. Each instruction in $\mathcal{A}$ is given two labels: a pre-label and a post-label. The labels must be distinct, except that for any two consecutive instructions the post-label of the first and the pre-label of the second must be identical, and for any labelled instruction, the pre-label and the label in the instruction must be identical.

For each location $l \in L^{\mathcal{A}}$ its mode can be derived from the transitions in $T^{\mathcal{A}}$. The set of transitions $T^{\mathcal{A}}$ is constructed by deriving one or more transitions from each instruction in $\mathcal{A}$:[7]

- An instruction `movl a, b` (move contents of `a` to `b`[8]) with pre-label l and post-label l' leads to a transition $(l, \texttt{true}, b := a, l')$.
- An instruction `jmp Lx` (unconditional jump to label `Lx`) with pre-label l leads to a transition $(l, \texttt{true}, \varepsilon, Lx)$.
- An instruction `cmpl a,b` (compare `a,b` and set zero flag if `a` = `b` and sign flag if `a` > `b`) with pre-label l and post-label l' leads to a transition $(l, \texttt{true}, (\texttt{ZF}, \texttt{SF}) := (a = b, a > b), l')$.
- An instruction `jle Lx` (conditional jump to `Lx`, jump if previous compare evaluated to "less or equal") with pre-label l and post-label l' leads to transitions $(l, \neg(ZF \vee SF), \varepsilon, l')$ and $(l, ZF \vee SF, \varepsilon, Lx)$.
- An instruction `l:incl i` (increment i by 1) with pre-label l and post-label l' leads to a transition $(l, \texttt{true}, (i, ZF, SF) := (i + 1, i = -1, i < -1), l')$. (We will ignore the assignments to ZF, SF in the following paragraphs and figures, since their values after increment instructions have no impact on the execution of $\mathcal{A}$.)

Further assembler instructions yield IOTS transitions in an analogous way. In the semantics we have ignored the overflow flag `OF`, as it can be proved that overflow will never happen. (The generated instructions that potentially could give overflow are of the form `l:incl i` coming from loop increments in $\mathcal{M}$, but the upper bound of these are some constants that are far smaller than the range values for `i+1`.)

7.6. Abstraction Mappings

When a SystemC model $\mathcal{M}$ is compiled into an assembler program $\mathcal{A}$, there is (as noted above) a 1-1 correspondence between SystemC symbols in $V^{\mathcal{M}}$ and assembler symbols in $V^{\mathcal{A}}$ (e.g. for each SystemC array element $x[n]$ there is a corresponding assembler array element $x(, n, 4)$), except that $V^{\mathcal{A}}$ contains additional local variables: flags, registers and stack addresses. We now define an IOTS $\mathcal{T}(\mathcal{M}^+)$ that extends the variable set of $\mathcal{T}(\mathcal{M})$ with local variable symbols corresponding to the additional flags, registers and stack addresses in $V^{\mathcal{A}}$: $\mathcal{T}(\mathcal{M}^+) = (V^{\mathcal{M}^+}, \sigma_0^{\mathcal{M}^+}, L^{\mathcal{M}^+}, T^{\mathcal{M}^+})$ where

- $V^{\mathcal{M}^+} = V_I^{\mathcal{M}^+} \cup V_O^{\mathcal{M}^+} \cup V_P^{\mathcal{M}^+}$.

[7] To make the description of this derivation easier to read, we use concrete syntax for assembler instructions, IOTS guards, and IOTS assignments.

[8] We use notational conventions of the GNU assembler; source operands are denoted on the left-hand side, target operands on the right-hand side.

- $V_I^{\mathcal{M}^+} = V_I^{\mathcal{M}}$ is the set of input variables.
- $V_O^{\mathcal{M}^+} = V_O^{\mathcal{M}}$ is the set of output variables.
- $V_P^{\mathcal{M}^+} = V_G^{\mathcal{M}^+} \cup V_L^{\mathcal{M}^+}$ is the set of processing variables.
- $V_G^{\mathcal{M}^+} = V_G^{\mathcal{M}}$.
- $V_L^{\mathcal{M}^+} = V_L^{\mathcal{M}} \cup CREGS \cup FLAGS \cup SADDR = (V_L^{\mathcal{A}} \setminus REGS) \cup CREGS$.
- $CREGS = \{\texttt{eax}, ...\}$.

We then define a map $\alpha^{\mathcal{M}}$ from $V^{\mathcal{A}}$ to $V^{\mathcal{M}^+}$:

- $\alpha^{\mathcal{M}}(\texttt{x(,n,4)}) = \texttt{x[n]}$ for the array elements $\texttt{x(,n,4)} \in V^{\mathcal{A}} \setminus V_L^{\mathcal{A}}$
- $\alpha^{\mathcal{M}}(\texttt{x}) = \texttt{x}$ for $\texttt{x} \in V_L^{\mathcal{A}} \setminus REGS$
- $\alpha^{\mathcal{M}}(\texttt{\%n}) = \texttt{n}$ for $\texttt{\%n} \in REGS$

Clearly $\alpha^{\mathcal{M}}$ is a bijection that preserves the mode of variables (input/output/processing), and its restriction $\alpha_{IO}^{\mathcal{M}} = \alpha^{\mathcal{M}}/(V_I^{\mathcal{A}} \cup V_O^{\mathcal{A}})$ to I/O variables is an *I/O map*.

7.7. Implementation Relation

Using the definitions given in the sections above, we are now able to define the implementation relation between assembler programs and SystemC models.

Definition 7.1. An assembler program $\mathcal{A} = g(\mathcal{M})$ is a *correct implementation* of a generated SystemC model $\mathcal{M}$, if $\mathcal{T}(\mathcal{A})$ is I/O equivalent to $\mathcal{T}(\mathcal{M})$ wrt. $\alpha_{IO}^{\mathcal{M}}$, i.e. $\text{equiv}(\mathcal{T}(\mathcal{A}), \mathcal{T}(\mathcal{M}), \alpha_{IO}^{\mathcal{M}})$.

In next section we explain our strategy for performing mechanised proofs of I/O equivalences. That strategy is based on the following theorem.

Theorem 7.1. An assembler program $\mathcal{A} = g(\mathcal{M})$ is a *correct implementation* of a generated SystemC model $\mathcal{M}$, if $\text{equiv}(\alpha^{\mathcal{M}}(\mathcal{T}(\mathcal{A})), \mathcal{T}(\mathcal{M}^+))$, where $\alpha^{\mathcal{M}}(\mathcal{T}(\mathcal{A}))$ is the result of renaming the variables in $\mathcal{T}(\mathcal{A})$ according to the map $\alpha^{\mathcal{M}}$.

Proof. This follows from the fact that $\text{equiv}(\mathcal{T}(\mathcal{A}), \mathcal{T}(\mathcal{M}), \alpha_{IO}^{\mathcal{M}})$ is true if and only if $\text{equiv}(\mathcal{T}(\mathcal{A}), \mathcal{T}(\mathcal{M}^+), \alpha_{IO}^{\mathcal{M}})$ (as adding processing variables to the state space of one IOTS, does not change the I/O behaviour of that IOTS) and the fact that $\text{equiv}(\mathcal{T}(\mathcal{A}), \mathcal{T}(\mathcal{M}^+), \alpha_{IO}^{\mathcal{M}})$ is true if and only if $\text{equiv}(\alpha^{\mathcal{M}}(\mathcal{T}(\mathcal{A})), \mathcal{T}(\mathcal{M}^+))$. □

7.8. Automated Object Code Verification

Currently an object code verifier is being implemented. The implementation consists of the following major components:

- An implementation of the two $\mathcal{T}$ functions yielding IOTS generators for SystemC and assembler code, respectively.
- A library of equivalence-preserving transformation rules similar to the ones exemplified in Fig. 5, 6.
- An IOTS transformer that given an IOTS and a transformation rule is able to apply the transformation rule.

A mechanised proof of the equivalence between an assembler program $\mathcal{A}$ and the SystemC controller model $\mathcal{M}$ from which it is generated is planned to be automatically performed according to the following procedure using the above components. First the SystemC controller model $\mathcal{M}$ is mapped to its behavioural IOTS model $\mathcal{T}(\mathcal{M})$, and $\mathcal{A}$ is mapped to its model $\mathcal{T}(\mathcal{A})$ as well, using the semantic evaluation functions for SystemC and assembler, respectively. Next, the symbols of $\mathcal{T}(\mathcal{A})$ are changed to C-style notation according to mapping $\alpha^{\mathcal{M}}$ defined in Sect. 7.6 – this results in $\mathcal{T}^1$. Also, the variable symbol space of $\mathcal{T}(\mathcal{M})$ is extended to $\mathcal{T}(\mathcal{M}^+)$, so that $\mathcal{T}^1$ and $\mathcal{T}(\mathcal{M}^+)$ can be directly compared with respect to their variable symbols. Then the verifier searches for a sequence $\mathcal{T}^1 \mapsto \mathcal{T}^2 \mapsto \ldots \mapsto \mathcal{T}(\mathcal{M}^+)$ of transformations from $\mathcal{T}^1$ to $\mathcal{T}(\mathcal{M}^+)$, whereupon it terminates.

7.9. Example

We illustrate the mechanised proof procedure explained above using a fragment of the SystemC controller code from our case study. Here global shadow variables `reqsigNext[i]` (the new state required for signal i) and `reqptNext[j]` (the new state required for point j) are copied to output signals `reqsig[i]` (set-state request to signal i) and `reqpt[j]` (set-state request to point j) during the output phase of a main loop cycle. Consider the following fragment from the output phase of a SystemC controller $\mathcal{M}$:

```
for ( int i=0; i<NUM_SIGNALS; i++)
  reqsig[i] = reqsigNext[i];
for ( int j=0; j<NUM_POINTS; j++)
  reqpt[j] = reqptNext[j];
```

The concrete configuration data for this controller instance defines `NUM_POINTS=3` and `NUM_SIGNALS=3`. From that the compiler[9] generates the following assembler fragment of $\mathcal{A}$:

```
  movl $0, i
  jmp  .L103
.L104:
  movl i, %edx
  movl i, %eax
  movl reqsigNext(,%eax,4), %eax
  movl %eax, reqsig(,%edx,4)
  movl i, %eax
  incl %eax
  movl %eax, i
.L103:
  movl i, %eax
  cmpl $2, %eax
  jle  .L104
  movl $0, j
  jmp  .L106
.L107:
  movl j, %edx
  movl j, %eax
  movl reqptNext(,%eax,4), %eax
  movl %eax, reqpt(,%edx,4)
  movl j, %eax
  incl %eax
  movl %eax, j
.L106:
  movl j, %eax
  cmpl $2, %eax
  jle  .L107
```

Now the mechanised equivalence proof is constructed as follows. (1) The behavioural IOTS model $\mathcal{T}(\mathcal{A})$ of $\mathcal{A}$ is constructed by using the semantic evaluation function for assembler programs. After changing the names of assembler variables to C-style notation according to mapping $\alpha^{\mathcal{M}}$ explained above, this results in an IOTS $\mathcal{T}^1$ which is depicted in Fig. 7. (2) The behavioural IOTS model $\mathcal{T}(\mathcal{M})$ of $\mathcal{M}$ is constructed by using the semantic evaluation function for SystemC models whereupon the variable space is extended to achieve $\mathcal{T}(\mathcal{M}^+)$ which is depicted on the left-hand side of Fig. 9. (3) Applying the transformation rule shown in Fig. 6 to the regions S_{11} and S_{12} of $\mathcal{T}^1$ results in an I/O-equivalent IOTS $\mathcal{T}^2$ depicted in Fig. 8. (4) Twofold application of other transformation rules on $\mathcal{T}^2$ results in I/O-equivalent IOTS $\mathcal{T}^3$ shown on the left-hand side of Fig. 9. (5) Finally, a valuation-preserving change of guard conditions ($[i \leq 2] \mapsto [i < 3], [i > 2] \mapsto [i \geq 3]$

[9] We have used `gcc 4.0.2` for this example.

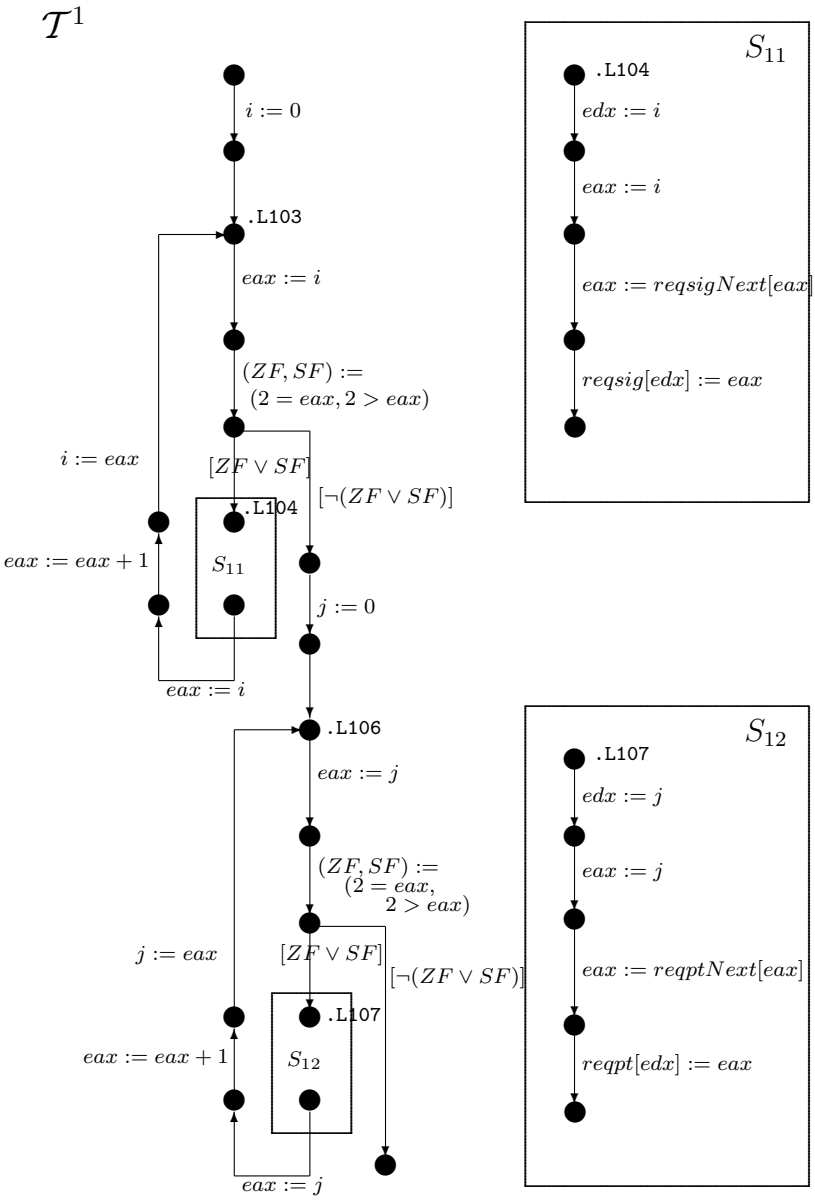

Fig. 7. IOTS $\mathcal{T}^1$ associated with $\mathcal{A}$ after renaming of variables.

etc.) yields $\mathcal{T}(\mathcal{M}^+)$ which completes the proof, as far as the code fragments shown here for illustration purposes are concerned.

8. Conclusion

In this paper we have given an overview of a complete model-driven development and verification approach for railway and tram control systems. The approach provides a framework consisting of:

1. A domain-specific language.
2. A collection of tools, including (a) syntax and static semantics checkers for the language, (b) generators producing executable models of the control system and its physical environment as well as proof obligations, (c) a bounded model checker, and (d) an object code verifier.

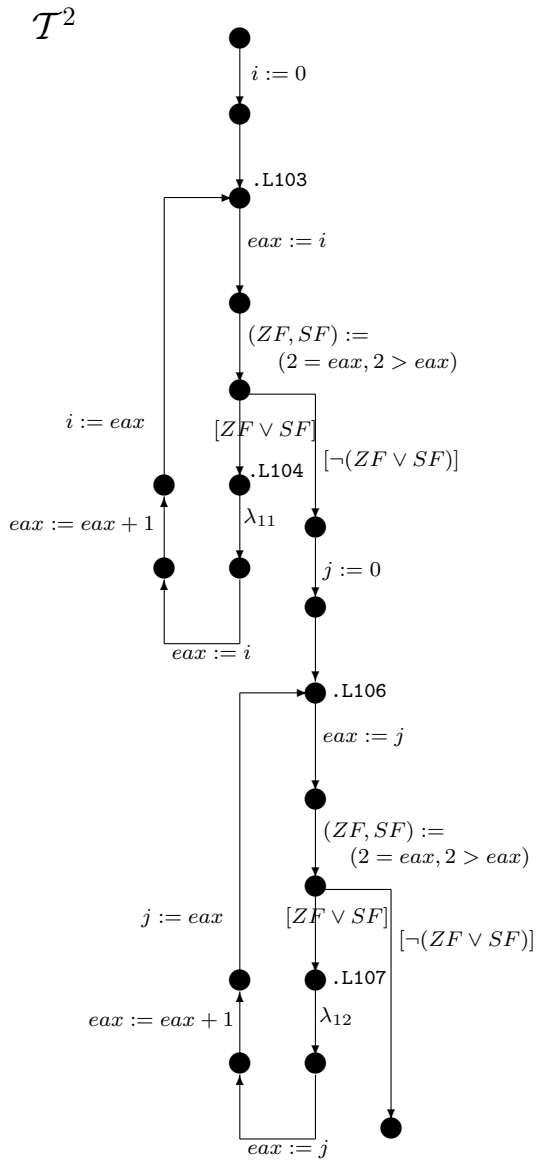

Fig. 8. $\mathcal{T}^1 \mapsto \mathcal{T}^2$: I/O-equivalent transformation

3. A method for using these tools to construct and verify a family of similar control systems.

For each control system to be generated, the user makes a description of the application-specific parameters in the domain-specific language and checks the description by means of the syntax and static semantics checker. Next, the generators produce models of the control system and its physical environment from this description, together with the safety requirements which are automatically verified using the bounded model checker in combination with an inductive proof strategy. Finally – since the formal controller model can be directly compiled – object code is generated by a conventional compiler, and it is checked by the object code verifier that the object code is behaviourally equivalent to the control system model. In this way it is ensured that the safety properties established for the control system model also hold for the object code.

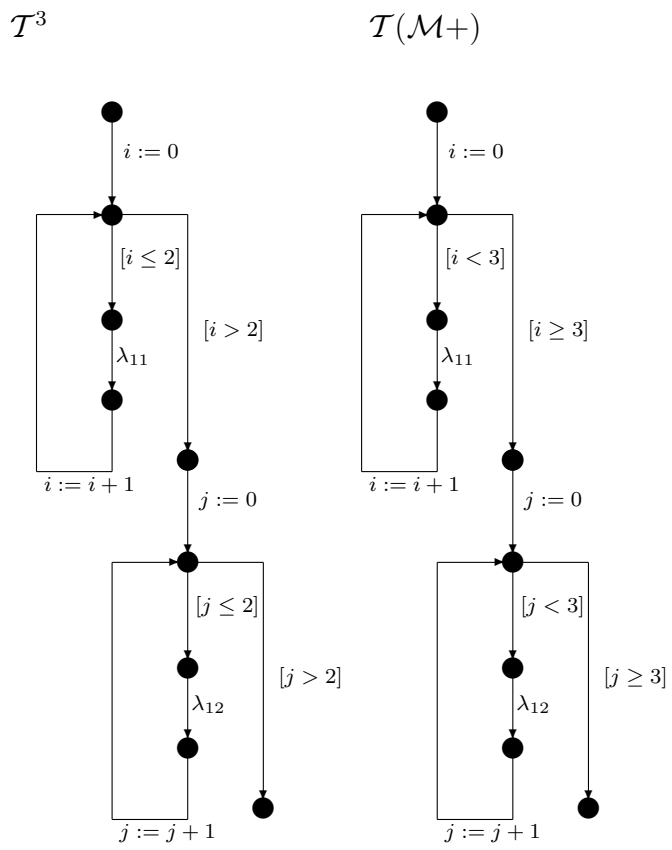

Fig. 9. $\mathcal{T}^3 \mapsto \mathcal{T}(\mathcal{M}^+)$: I/O-equivalent guard transformation.

The development of the framework was formalised by using the RAISE formal method, thereby providing complete and precise specifications of the tools as well as the domain-specific language. This provides a sound basis for tool implementation and allows for formal mechanised verification of algorithms.

References

[Acc04] Accellera. *Property Specification Language Version 1.1*, 2004.

[Ber06] K. Berkenkötter. OCL-based validation of a railway domain profile. In *OCLApps 2006 - OCL for (Meta-)Models in Multiple Application Domains*, October 2006.

[BFPT06] B. Badban, M. Fränzle, J. Peleska, and T. Teige. Test automation for hybrid systems. In *Proceedings of the Third International Workshop on SOFTWARE QUALITY ASSURANCE (SOQUA 2006)*, Portland Oregon, USA, November 2006.

[BGH+97] D. Bjørner, C.W. George, B. Stig Hansen, H. Laustrup, and S. Prehn. A railway system, coordination'97, case study workshop example. Technical Report 93, UNU/IIST, P.O.Box 3058, Macau, 1997.

[Bjø03a] D. Bjørner. Domain Engineering: A "Radical Innovation" for Software and Systems Engineering? A Biased Account. In Nachum Dershowitz, editor, *The Zohar Manna Intl.Symp. on "Verification: Theory & Practice"*, Heidelberg, Germany, July 2003. Springer-Verlag.

[Bjø03b] D. Bjørner. New Results and Current Trends in Formal Techniques for the Development of Software for Transportation Systems. In *Proceedings of the Symposium on Formal Methods for Railway Operation and Control Systems (FORMS'2003), Budapest/Hungary*. L'Harmattan Hongrie, May 15-16 2003.

[Bjø03c] D. Bjørner. Railways systems: Towards a domain theory. Technical report, Informatics and Mathematical Modelling, Technical University of Denmark, Building 322, Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark, 2003.

[Bjø06a] D. Bjørner. *Software Engineering, vol. 1: Abstraction and Modelling*. Texts in Theoretical Computer Science. Springer, 2006.

[Bjø06b] D. Bjørner. *Software Engineering, vol 2: Specification of Systems and Languages*. Texts in Theoretical Computer Science. Springer, 2006.

[Bjø06c] D. Bjørner. *Software Engineering, vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science. Springer, 2006.

[Bjø06d] D. Bjørner. The Rôle of Domain Engineering in Software Development, October 2006. Invited keynote paper and talk: IPSJ/SIGSE Software Engineering Symposium 2006, Tokyo.

[Bjø07] D. Bjørner. Domain Engineering, August 2006, reprinted March 2007. To appear as a chapter in a book based on the BCS FACS Evening Seminars to be published by Springer (UK).

[DC04] R. Dyhrberg and N. Christensen. A Domain-Specific Language for Tramway Control Systems. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, May 2004.

[DG05] R. Drechsler and D. Große. System level validation using formal techniques. *IEE Proc.-Comput. Digit. Tech.*, 152(3):393–406, May 2005.

[ECfES01] European Committee for Electrotechnical Standardization. *EN 50128 – Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems*. CENELEC, Brussels, 2001.

[EDD+04] H. Ehrig, W. Damm, J. Desel, M. Große-Rhode, W. Reif, E. Schnieder, and E. Westkämper, editors. *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*. Springer Verlag, 2004. ISBN 3-540-23135-8.

[GH03] T. Gjaldbæk and A. E. Haxthausen. Modelling and Verification of Interlocking Systems for Railway Lines. In *Proceedings of the 10th IFAC Symposium on Control in Transportation Systems*. Elsevier Science Ltd, Oxford, 2003. ISBN 0-08-044059-2.

[GLMS02] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[GZ99] G. Goos and W. Zimmermann. Verification of compilers. In *Correct System Design*, pages 201–230. Springer, 1999.

[HCD04] A. E. Haxthausen, N. Christensen, and R. Dyhrberg. 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*, 2004.

[HP00a] A. E. Haxthausen and J. Peleska. Formal Development and Verification of a Distributed Railway Control System. *IEEE Transaction on Software Engineering*, 26(8):687–701, 2000.

[HP00b] A. E. Haxthausen and J. Peleska. 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*, pages 263–271. VDI-Verlag, Düsseldorf, 2000.

[HP02] A. E. Haxthausen and J. Peleska. 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*, June 23-28 2002.

[HP03a] A. E. Haxthausen and J. Peleska. 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, 2003.

[HP03b] A. E. Haxthausen and J. Peleska. 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), Budapest/Hungary*, pages 83–90. L'Harmattan Hongrie, May 15-16 2003.

[LVH00] M. P. Lindegaard, P. Viuf, and A. E. Haxthausen. Modelling Railway Interlocking Systems. In *Proceedings of the 9th IFAC Symposium on Control in Transportation Systems 2000, June 13-15, 2000, Braunschweig, Germany*, pages 211–217, 2000.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[MRR03] W. Müller, J. Ruf, and W. Rosenstiel. *SystemC – Methodologies and Applications*, chapter 4, pages 97–126. Kluwer Academic Publishers, 2003.

[PBH00] J. Peleska, A. Baer, and A. E. Haxthausen. 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*, pages 147–152, 2000.

[PGHD04] J. Peleska, D. Große, A. E. Haxthausen, and R. Drechsler. Automated verification for train control systems. In E. Schnieder and G. Tarnai, editors, *Proceedings of the FORMS/FORMAT 2004 - Formal Methods for Automation and Safety in Railway and Automotive Systems*, pages 252–265. Technical University of Braunschweig, December 2004. ISBN 3-9803363-8-7.

[PH07] J. Peleska and A. E. Haxthausen. Object Code Verification for Safety-Critical Railway Control Systems. In *Proceedings of Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2007), Braunschweig, Germany*. GZVB e.V., 2007. ISBN 13:978-3-937655-09-3.

[PSS98] A. Pnueli, O. Shtrichman, and M. Siegel. The code validation tool CVT: Automatic verification of a compilation process. *International Journal on Software Tools for Technology Transfer*, 2(2):192–201, 1998.

[RAI92] The RAISE Language Group. *The RAISE Specification Language*. The BCS Practitioners Series. Prentice Hall Int., 1992.

[RJB04] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language – Reference Manual, 2nd edition*. Addison-Wesley, July 2004.

[ST04] E. Schnieder and G. Tarnai, editors. *Proceedings of Formal Methods for Automation and Safety in Railway and*

Automotive Systems (FORMS/FORMAT 2004)), Braunschweig, Germany. Technical University of Braunschweig, December 2004.

[ST07] E. Schnieder and G. Tarnai, editors. *Proceedings of Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2007)), Braunschweig, Germany*. GZVB e.V., 2007. ISBN 13:978-3-937655-09-3.

[TS03] G. Tarnai and E. Schnieder, editors. *Proceedings of the Symposium on Formal Methods for Railway Operation and Control Systems (FORMS'2003)*, Budapest, 2003. L'Harmattan Hongrie.

[W3Ca] XForms 1.0. Available under http://www.w3.org/TR/xforms.

[W3Cb] Extensible Markup Language (XML). Available under http://www.w3.org/XML/.

[W3Cc] The Extensible Stylesheet Language Family (XSL). Available under http://www.w3.org/Style/XSL.