OBJECT CODE VERIFICATION FOR SAFETY-CRITICAL RAILWAY CONTROL SYSTEMS

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

Abstract: In this article we describe a method for fully automated object code verification, applicable to railway control systems developed within a framework previously proposed by the authors. This allows us to apply arbitrary off-the-shelf compilers in a safety-critical context without having to perform expensive compiler validations. Within the restrictions of the framework, the object code verification is less complex than the general problem which has been already been investigated by other authors. Therefore it can be performed quite efficiently: High-level code $\mathcal{M}$ written in SystemC, C or C++ and the associated assembler code $\mathcal{A}$ generated by the compiler are both lifted to transition system models $\mathcal{T}(\mathcal{M}), \mathcal{T}(\mathcal{A})$, respectively, representing their behaviour. A generic theory containing equivalence preserving transformations on transition systems is elaborated and proven. Using a pattern matching system on these behavioural models, the transformations are applied with a strategy to transform $\mathcal{T}(\mathcal{M})$ into $\mathcal{T}(\mathcal{A})$ or vice versa. If the transformation succeeds, this establishes behavioural equivalence between $\mathcal{M}$ and $\mathcal{A}$.

Keywords: Verification tools, object code verification, railway control systems

1. INTRODUCTION

Motivation. In this article, we describe a method for fully automated object code verification, applicable for railway control systems developed within a development framework previously proposed by the authors. The work presented in this article represents new results obtained within a joint effort of several authors in the field of domain-specific description formalisms, model-based development, formal verification and automated HW/SW integration testing for embedded railway control systems, previously published in (Peleska *et al.*, 2004; Haxthausen and Peleska, 2000; Haxthausen and Peleska, 2002; Berkenkötter, 2006) and further references given in these publications.

Automated object code verification for railway control systems is motivated by the fact that applicable standards for these safety-critical applications, in particular (European Committee for Electrotechnical Standardization, 2001), require a substantial justification with respect to the consistency between high-level software code (e. g. C/C++ programs) and the object code generated by the associated compilers. The conventional way for this is to *validate* the compiler. This validation, however, is very time-consuming and has to be performed again whenever modifications of the compiler – such as optimisations or the introduction of new pragmas – have been performed. Moreover, these validations are still far from being formal proofs, so errors may still be present in validated compilers. Strategies for fully formal compiler verification have been elaborated by several authors – see (Goos and Zimmermann, 1999) and the references listed there. According to our knowledge, however, formally verified compilers for the development of railway control systems of highest criticality level *(Safety Integrity Level SIL-4)* (European Committee for Electrotechnical Standardization, 2001) are currently not used in practice.

The object code verification approach is complementary to compiler verification: An arbitrary compiler is used, and instead of a compiler validation, the generated object code is verified against its "specification", that is, the associated high-level code. At first glance, object code verification has a draw-back when compared to compiler verification: While the latter is done "once and for all", object code verification has to be performed whenever a newly compiled program is "finalised" to become operational in the target environment. But industrial practice shows that compiler updates occur quite frequently, so that a considerable number of non-trivial re-verifications have to be performed. In contrast to that, object code verification can be fully automated and reasonably fast, if the compiled code originates from high-level programs strictly adhering to certain programming patterns. While this assumption is certainly not true for arbitrary, manually written, and, in particular, object-oriented code, observation of these patterns can be easily enforced for high-level source code generated from abstract specification models in an automatic way.

Related Work. Code validation – that is, the investigation of implementation relations between high-level and low-level programming code – has been investigated by several authors, ssee (Pnueli *et al.*, 1998) 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 introduced in Section 4 is a specialisation of the "correct implementation relation" defined in (Pnueli *et al.*, 1998) – we exploit the specific restrictions of our model-based development framework in order to simplify the equivalence proofs in a considerable way. In particular, our restriction to integer data types – which is possible and suitable for the railway domain but would not be acceptable in other application areas – and the utilisation of a simple programming model (Section 3) lead to mechanised proof techniques which can be based on pattern matching and an associated generic theory describing equivalence preserving transformations for these patterns (Section 4).

The I/O transition systems serving as behavioural models in this article have a rather wide application range, see (Badban *et al.*, 2006) for an example from the field of automated hybrid systems testing.

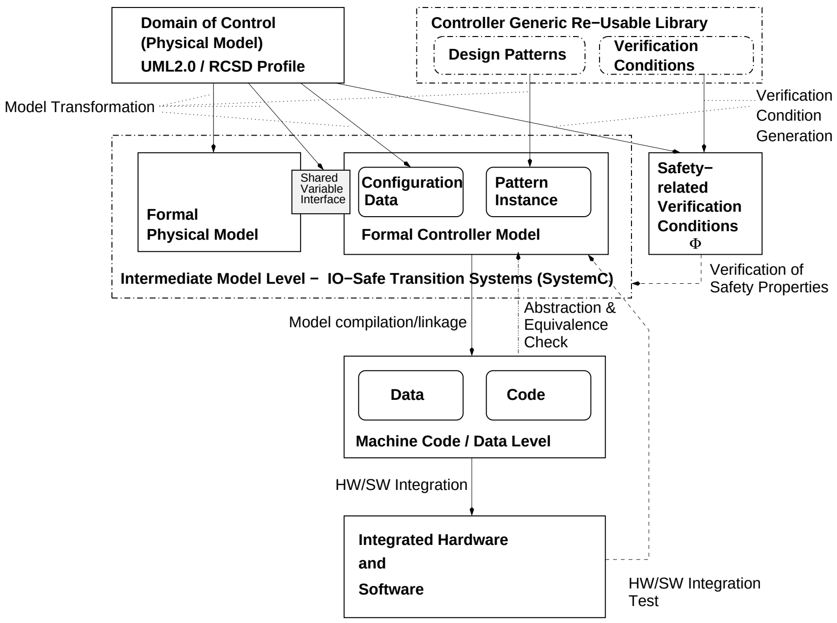
Paper Overview. In order to make this article sufficiently self-contained, we give an overview of the complete model-based development and verification approach in the context of which the object code verification is taking place in Section 2. In Section 3 we introduce the controller model from which the object code is generated. Furthermore, in Section 4, we introduce some theoretical foundations needed for the object code verification. Then in Section 5 a description of our method for object code verification is given. Section 6 presents the conclusion.

2. MODEL DEVELOPMENT AND VERIFICATION APPROACH

Our general strategy for model-driven development and automated verification and testing of railway control systems is to go through the following steps illustrated in Figure 1:

1. Domain-specific model creation and verification: The railway specialists describe domain-specific details (like station geography and train routes) of the system to be developed in a domain-specific formalism which is encoded as a UML 2.0 profile (Haxthausen and Peleska, 2002; Berkenkötter, 2006). The profile has a formally defined static semantics, so that each concrete model $\mathcal{D}$ – expressed as an object diagram generated within the rules of the profile – can be verified against constraints specifying the rules for models which are admissible with respect to syntax and static semantics.

2. Controller model generation: The domain-specific model $\mathcal{D}$ is automatically transformed into



Domain of Control
(Physical Model)
UML2.0 / RCSD Profile
Controller Generic Re−Usable Library
Design Patterns
Verification Conditions
Model Transformation
Verification Condition Generation
Formal Physical Model
Shared Variable Interface
Configuration Data
Pattern Instance
Formal Controller Model
Safety−related Verification Conditions Φ
Intermediate Model Level − IO−Safe Transition Systems (SystemC)
Verification of Safety Properties
Model compilation/linkage
Abstraction & Equivalence Check
Data
Code
Machine Code / Data Level
HW/SW Integration
Integrated Hardware and Software
HW/SW Integration Test

Fig. 1. Model-driven development and verification steps.

a behavioural *controller model* $\mathcal{M}$ with transition system semantics, expressed in SystemC (Peleska *et al.*, 2004). To be more specific, the controller model $\mathcal{M}$ is a generic design pattern instantiated with configuration data obtained from the domain-specific model $\mathcal{D}$.

3. Controller model verification: Together with the controller model $\mathcal{M}$, a behavioural *physical model* $\mathcal{P}$ (describing in SystemC how physical devices like signals are behaving) and a set of *verification obligations* Φ are generated. Next it is verified that the verification obligations Φ imply that the controller model $\mathcal{M}$ in concurrent combination with the physical model $\mathcal{P}$ is guaranteed to fulfil its safety properties (as, for example, the requirement that trains should never meet within a track segment or on a point). Finally, the obligations Φ are discharged by means of an inductive proof strategy, where the induction step is performed using bounded model checking techniques (Peleska *et al.*, 2004).

4. Code generation: The controller model $\mathcal{M}$ and its configuration data are compiled into object code and data with conventional C/C++ compilers. This results in an assembler "model" $\mathcal{A}$. Since the SystemC model conforms to a number of coding patterns and interface restrictions which can be easily enforced on typical controller hardware and since compiler optimisations are not used for safety-critical applications, $\mathcal{A}$ also adheres to a set of simple "low-level patterns".

5. Code verification: The assembler "model" $\mathcal{A}$ is verified to be behavioural equivalent to the controller model $\mathcal{M}$. Section 5 explains how that is done.

6. Hardware/Software Integration and Test: Finally the correctness of the hardware/software integration is automatically tested, following the concepts described in (Badban *et al.*, 2006).

3. CONTROLLER MODEL

In this section we give a short overview of the controller model $\mathcal{M}$ specified using SystemC and explain it behaviour in an intuitive way. The precise behavioural model semantics will be introduced in Section 5 below. More details about the model and its associated safety verification have been described in (Peleska *et al.*, 2004).

Interfaces. Controller $\mathcal{M}$ monitors the outputs of track elements whose state is updated and relayed to $\mathcal{M}$ by means of shared variable arrays (located, for example, in a DMA area where the interface controller may place the data), such as `actsig[i]` (the actual state of signal i, encoded by integral numbers representing HALT and GO), `actpt[j]` (actual state of point j) , `actsens[k]` (the actual state of sensor k, indicating HI if a train passes and LO otherwise), and `req[r]` (request of a train to enter route r). The controller acts on signals and points by sending control commands on output arrays `reqsig[i]` (requested – potentially new – state of signal i) and `reqpt[j]` (requested state of point j).

Behaviour. The main task of the controller is to switch points and signals in such a way that route requests of trains are granted without impairing the safety of the railway network. This is achieved by (1) keeping track of assigned routes, using internal state variables of $\mathcal{M}$, (2) evaluating conflicts of requested routes with others currently assigned, (3) monitoring the consistency between requested and actual state of track elements and (4) by setting the points associated with an assigned route appropriately before granting it to a train by setting the associated signal to GO.

Programming Model. The SystemC controller models $\mathcal{M}$ are implemented using a main loop, so that each execution cycle has four phases: In the *input phase* all current values of input signals are copied to (global) shadow variables, in the *processing phase* signals 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, and in the *output phase* the states of global variables shadowing outputs are copied to the corresponding output signals. The objective of the wait phase is to ensure constant loop frequency

$$f = \frac{1}{\Delta(in)+\Delta(proc)+\Delta(wait)+\Delta(out)}[Hz]$$

The durations $\Delta(in)$ and $\Delta(out)$ of input and output phases, respectively, may be assumed to be constant values, since on machine level they consist of a constant number of address calculations and move operations (caching effects can be neglected). The duration of the processing phase, however, may vary in each loop, so that the objective of the wait period is to ensure an approximately constant value for $const = \Delta(proc) + \Delta(wait)$. An excessive duration of $\Delta(proc) > const$ would lead to a fail-safe halt of the system, so that we can rely on the system to either process main loop cycles in constant time or to stop. The wait phase is implemented using the *time stamp counter* of the CPU. For the remainder of this article this phase will ignored.

4. THEORETICAL FOUNDATIONS

In this section we will introduce *I/O-safe transitions systems (IOTS)*, an associated equivalence relation and a generic theory containing theorems on equivalence-preserving transformations of IOTS. As will be shown in Section 5 below, IOTS serve as behavioural models for both SystemC programs and their associated assembler programs. By application of the generic theory, I/O-equivalence between SystemC and assembler programs can be established in a straightforward way.

4.1 I/O-Safe Transition Systems

An IOTS is defined as a quadruple $\mathcal{T} = (Loc, Init, V, Trans)$. The set of *locations* Loc is partitioned into pairwise disjoint subsets $Loc = Loc_I \cup Loc_O \cup Loc_P$ called input, output and processing locations, respectively. The *variable space* V is partitioned into mutually disjoint subsets $V = V_I \cup V_O \cup V_P$ called input, output and processing variables, respectively. V_P is further partitioned into mutually disjoint subsets $V_P = V_L \cup V_G \cup V_C$ called local and global variables and global constants. *Init* $\in$ *Loc* denotes the *initial location* of $\mathcal{T}$. The set *Trans* of *transitions* consists of quadruples from $Loc \times Guard \times Assign \times Loc$ where *Guard* is the set of *transition guards* and consists of all quantifier-free predicates over $V_P \cup \{\mathtt{false}, \mathtt{true}\}$. Observe that this implies that input or output variables never occur in guard conditions. The set of *assignments Assign* is defined as the collection of all pairs $\vec{x} := \vec{t}$ where $\vec{x}$ is a vector of variables from $V - V_I$ and $\vec{t}$ is a vector of terms over V with the same dimension as $\vec{x}$. Now *input locations* $l \in Loc_I$ are distinguished by the fact that all transitions entering l have assignments that only read from input variables $x \in V_I$ and only change locals and globals from $V_G \cup V_L$. *Processing locations* $l \in Loc_P$ have incoming transitions with guards and assignments that only read processing variables (from V_P) and only change global and local variables (from $V_G \cup V_L$). *Output locations* $l \in Loc_O$ have incoming transitions with guards and assignments only reading from V_P and assigning to V_O.

As a consequence of these definitions, when running (see below) the IOTS $\mathcal{T}$, concurrent changes of input variables do not affect the enabling (evaluation of guards g) and effect (of assignments $\vec{x} := \vec{t}$) of transitions $(l, g, \vec{x} := \vec{t}, l')$ into processing and output locations (i.e. when $l' \in Loc_O \cup Loc_P$.)

A *run* of $\mathcal{T}$ is a sequence of pairs $\langle (l_0, \sigma_0), (l_1, \sigma_1) \ldots \rangle$ of locations and valuations $\sigma_i : V \to D$ which satisfies the properties (1) $l_0 = Init$ and (2) $\forall\, i \geq 0 : \exists (l, g, \vec{x} := \vec{t}, l') \in Trans : l = l_i, l' = l_{i+1}, \sigma_i(g) = \mathtt{true}, \sigma_{i+1}(\vec{x}) = \sigma_i(\vec{t})$, where $\sigma_i(g)$, $\sigma_i(\vec{x})$ and $\sigma_i(\vec{t})$ are the canonic extensions of valuations to guard expressions, vectors and terms, respectively. Since IOTS never assign to input variables, the σ_i are undetermined with respect to input variable valuations. This models the behaviour of an unrestricted system environment which may place any input value at any time. Let $Run(\mathcal{T})$ denote the set of all runs of IOTS $\mathcal{T}$.

An IOTS $\mathcal{T}$ is called *deterministic* if in every possible run $\langle (l_0, \sigma_0), (l_1, \sigma_1) \ldots \rangle \in Run(\mathcal{T})$ only one transition is enabled at a time:

$$\begin{aligned} &\forall\, i \in \mathbb{N}_0 : \\ &\forall (l_i, g, a, l), (l_i, g', a', l') \in Trans : \\ &\qquad \sigma_i(g) = \mathtt{true} \wedge \sigma_i(g') = \mathtt{true} \Rightarrow \\ &\qquad\qquad (l_i, g, a, l) = (l_i, g', a', l') \end{aligned}$$

Since IOTS will serve as behavioural models for deterministic programs we will restrict ourselves to deterministic IOTS in this paper.

4.2 I/O-Equivalence

Given a run $r \in Run(\mathcal{T})$, we define r_{IO} as the restriction of r to input and output locations, and restrict each valuation σ occurring in such a location to $\sigma \mid_{(V_I \cup V_O)}$. Let $\rho : V_I^1 \cup V_O^1 \to V_I^2 \cup V_O^2$ be a bijective mapping between I/O variables of two IOTS $\mathcal{T}^1, \mathcal{T}^2$ such that input variables are mapped to inputs and outputs to outputs (that is, $\rho(V_I^1) = V_I^2$ and, since ρ is one-one and onto, also $\rho(V_O^1) = V_O^2$). Given ρ we can define I/O equivalence between runs r^1 of $\mathcal{T}^1$ and r^2 of $\mathcal{T}^2$: We write $r^1 \sim_\rho r^2$ if their restrictions

$$\begin{aligned} &r_{IO}^i = \langle (l_j^i, \sigma_j^i \mid_{(V_I^i \cup V_O^i)}) \mid \\ &\qquad\qquad\qquad 0 \leq j < \#r_{IO}^i \rangle \\ &i = 1, 2 \end{aligned}$$

satisfy the following conditions: (1) Both restrictions have the same length, written $\#r_{IO}^1 = \#r_{IO}^2$, (2) input/output locations occur in the same order, that is,

$$\forall\, 0 \leq j < \#r_{IO}^1 : l_j^1 \in Loc_I^1 \Leftrightarrow l_j^2 \in Loc_I^2$$

and (3) input and output valuations are ρ-equivalent in the sense that

$$\begin{aligned} &\forall\, 0 \leq j < \#r_{IO}^1 : \\ &\qquad \sigma_j^1 \mid_{(V_I^1 \cup V_O^1)} = \sigma_j^2 \circ \rho \end{aligned}$$

We can now define the ρ-equivalence of $\mathcal{T}^1, \mathcal{T}^2$ (written $\mathcal{T}^1 \sim_\rho \mathcal{T}^2$) by requiring a bijective mapping γ between runs of $\mathcal{T}^1$ and $\mathcal{T}^2$ such that

$$\forall\, r^1 \in Run(\mathcal{T}^1) : \gamma(r^1) \sim_\rho r^1$$

Since ρ and γ are required to be bijective it is trivial to see that $\sim_\rho$ is an equivalence relation for $\rho = id$ (the identity on $V_I \cup V_O$). In this case we drop the suffix *id* and write $\mathcal{T}^1 \sim \mathcal{T}^2$ instead of $\mathcal{T}^1 \sim_{id} \mathcal{T}^2$). For arbitrary bijections ρ_1, ρ_2 we have a generalised transitivity property in the sense that

$$\mathcal{T}^1 \sim_{\rho_1} \mathcal{T}^2 \wedge \mathcal{T}^2 \sim_{\rho_2} \mathcal{T}^3 \\ \Rightarrow \mathcal{T}^1 \sim_{\rho_2 \circ \rho_1} \mathcal{T}^3$$

4.3 Generic Theory on I/O-Equivalence

In this section theorems on IOTS transformations preserving I/O-equivalence will be established. These theorems are *generic* in the sense that they are universally quantified over variable names, guard conditions and location names and over the occurrence of certain *patterns* of locations and associated transitions; hence our utilisation of the term 'generic theory' for the collection of these theorems.

In the theorems an corollaries below, we refer to IOTS $\mathcal{T}^i = (Loc^i, Init^i, V^i, Trans^i), i = 1, 2$.

Theorem 1 *The following statement is generic in locations l_0, l_1, variable symbols x, y, i, aux_0, aux_1 and guards g.*

Let $\tau_0 = (l_0, g, x[i] := y[i], l_1) \in Trans^1$. Assume that on all $\mathcal{T}^1$-paths emanating from l_1 $aux_i \in V_L^1, i = 0, 1$ are first written to in an assignment before being used in a guard or a right-hand side assignment term. Then, using fresh location symbols $l_{0,j} \notin Loc^1, j = 0, 1, 2$, construct $\mathcal{T}^2$ as follows:
(1) $Loc^2 = Loc^1 \cup \{l_{0,0}, l_{0,1}, l_{0,2}\}$,
(2) $Init^2 = Init^1$, (3) $V^2 = V^1$,
(4) $Trans^2 = (Trans^1 - \{\tau_0\}) \cup T^2$ with

$$\begin{aligned} T^2 = \{&(l_0, g, aux_0 := i, l_{0,0}),\\ &(l_{0,0}, \texttt{true}, aux_1 := i, l_{0,1}),\\ &(l_{0,1}, \texttt{true}, aux_1 := y[aux_1], l_{0,2}),\\ &(l_{0,2}, \texttt{true}, x[aux_0] := aux_1, l_1)\}\end{aligned}$$

Then $\mathcal{T}^1 \sim \mathcal{T}^2$.

Proof. Given $r^1 \in Run(\mathcal{T}^1)$, we will construct $r^2 \in Run(\mathcal{T}^2)$ such that $r^1 \sim r^2$. To this end, r^1 is partitioned into

$$r^1 = u_0^1 \frown w_0^1 \frown u_1^1 \frown w_1^1 \frown \ldots$$

such that the location-valuation pairs in u_i^1 never visit l_0, whereas the segments $w_j^1 = \langle\, (l_0, \sigma_{j,0}), (m_j, \sigma_j)\,\rangle$ start in location l_0 and perform transitions into any post-location of l_0. In particular, l_1 is such a possible post-location m, and it is possible that several consecutive $w_j^1, w_{j+1}^1, \ldots$ occur in r^1 if the post-location of m^j is again l_0 (a u_j^1 segment may be empty). Our goal is to construct segments w_2^j such that

$$r^2 = u_0^1 \frown w_0^2 \frown u_1^1 \frown w_1^2 \frown \ldots$$

is in $Run(\mathcal{T}_2)$ and I/O-equivalent to r_1: Since u_0^1 does not visit location l_0, it can be performed by $\mathcal{T}_2$ as well. It remains to show the existence of w_0^2 such that $u_0^1 \frown w_0^1$ leaves $\mathcal{T}_1$ in the same state as $u_0^1 \frown w_0^2$ leaves $\mathcal{T}_2$. This existence is shown independently of the position j of w_j^2, so that the equivalence $r^1 \sim r^2$ follows by induction.

To construct w_0^2, two cases are distinguished for $w_0^1 = \langle(l_0, \sigma_{0,0}), (m_0, \sigma_0)\rangle$:

Case 1 – $\sigma_{0,0}(g) = \texttt{false} \vee m_0 \neq l_1$: In this situation, the transition τ associated with w_0^1 cannot be τ_0. As a consequence, $\tau \in Trans^2$ by construction $\mathcal{T}_2$, so we can choose $w_0^2 = w_0^1$ and there is nothing more to prove.

Case 2 – $\sigma_{0,0}(g) = \texttt{true} \wedge m_0 = l_1$: Since $\mathcal{T}^1$ is deterministic, w_0^1 must be the effect of transition τ_0. As a consequence,

$$\begin{aligned}(m_0, \sigma_0) = (&l_1, \sigma_{0,0} \oplus \\ &\{x[\sigma_{0,0}(i)] \mapsto \sigma_{0,0}(y[\sigma_{0,0}(i)])\})\end{aligned}$$

We define

$$\begin{aligned} w_0^2 = \langle&(l_0, \sigma_{0,0}), (l_{0,0}, \sigma_{0,1}),\\ &(l_{0,1}, \sigma_{0,2}), (l_{0,2}, \sigma_{0,3})(l_1, \sigma_{0,4})\rangle\end{aligned}$$

and set valuations $\sigma_{0,i}, i = 1, 2, 3, 4$ so that they reflect the effect of the new transitions introduced in T^2:

$$\begin{aligned}
\sigma_{0,1} &= \sigma_{0,0} \oplus \{aux_0 \mapsto \sigma_{0,0}(i)\},\\
\sigma_{0,2} &= \sigma_{0,0} \oplus \{aux_0 \mapsto \sigma_{0,0}(i),\\
&\qquad aux_1 \mapsto \sigma_{0,0}(i)\},\\
\sigma_{0,3} &= \sigma_{0,0} \oplus \{aux_0 \mapsto \sigma_{0,0}(i),\\
&\qquad aux_1 \mapsto \sigma_{0,0}(y[\sigma_{0,0}(i)]\}\\
\sigma_{0,4} &= \sigma_{0,0} \oplus \{aux_0 \mapsto \sigma_{0,0}(i),\\
&\qquad aux_1 \mapsto \sigma_{0,0}(y[\sigma_{0,0}(i)],\\
&\qquad x[\sigma_{0,0}(i)] \mapsto \sigma_{0,0}(y[\sigma_{0,0}(i)])\}
\end{aligned}$$

Now w_0^2 is well-defined by definition of T^2 and the $\sigma_{0,j}$. Moreover, $\sigma_0 \mid_X = \sigma_{0,4} \mid_X$ with $X = V^1 - \{aux_0, aux_1\}$. Since $aux_i \in V_L^1$ and since each of the new locations has only one entering transition, the $l_{0,0}, l_{0,1}, l_{0,2}$ are processing locations, so $u_0^1 \frown w_0^1$ and $u_0^1 \frown w_0^2$ still traverse the same sequence of input/output locations, and all valuations of I/O variables are identical in both sequences. Finally, since post-states of l_1 always write to aux_i before reading them, the differing valuations of σ_0 and $\sigma_{0,4}$ on $\{aux_0, aux_1\}$ do not have any effect. As a consequence, $u_0^1 \frown w_0^1 \sim u_0^1 \frown w_0^2$ and both runs can be extended in an equivalent way, either by a non-empty u_1^1 or by w_1^1 and an analogously constructed w_1^2. Since this process can be extended inductively, this establishes the existence of r^2 with $r^1 \sim r^2$.

Conversely, given $r^2 \in Run(\mathcal{T}^2)$, an equivalent run $r^1 \in Run(\mathcal{T}^1)$ is constructed as follows: r^2 is partitioned as defined above. Only the $w_j^2 = \langle (l_0, \sigma_{j,0}), \dots \rangle$ with $\sigma_{j,0}(g) = \texttt{true}$ have to be considered. Since $\mathcal{T}^1$ and therefore, by construction, also $\mathcal{T}^2$ are deterministic, w_j^2 must traverse $l_0, l_{0,0}, l_{0,1}, l_{0,2}, l_1$ in this situation. Since the aux_i are first written to from variables whose valuations are identical in r^2 and r^1 at corresponding locations, the existence of a fragment $w_j^1 = \langle (l_0, \sigma_{j,0}), (l_1, \sigma_j) \rangle$ with $\sigma_j \mid_X = \sigma_{0,4} \mid_X$ follows. This completes the proof. □

The following corollaries are immediate consequences from Theorem 1 and its proof, when simplifying the above theorem to variables x, y without indexes.

Corollary 1 *The following statement is generic in locations l_0, l_1, variable symbols x, y, aux_0 and guards g.*
Let $\tau_0 = (l_0, g, x := y, l_1) \in Trans^1$. Assume that on all $\mathcal{T}^1$-paths emanating from l_1 $aux_0 \in V_L^1$ is first written to in an assignment before being used in a guard or a right-hand side assignment term. Then, using a fresh location symbol $l_{0,0} \notin Loc^1$, construct $\mathcal{T}^2$ as follows:
(1) $Loc^2 = Loc^1 \cup \{l_{0,0}\}$,
(2) $Init^2 = Init^1$, (3) $V^2 = V^1$,
(4) $Trans^2 = (Trans^1 - \{\tau_0\}) \cup T^2$ with

$$T^2 = \{(l_0, g, aux_0 := y, l_{0,0}), \\ (l_{0,0}, \texttt{true}, x := aux_0, l_1)\}$$

Then $\mathcal{T}^1 \sim \mathcal{T}^2$. □

Corollary 2 *The following statement is generic in locations l_0, l_1, variable symbols x, y, aux_0 and guards g.*
Let $\tau_0 = (l_0, g, x := x + 1, l_1) \in Trans^1$. Assume that on all $\mathcal{T}^1$-paths emanating from l_1 $aux_0 \in V_L^1$ is first written to in an assignment before being used in a guard or a right-hand side assignment term. Then, using fresh location symbols $l_{0,j} \notin Loc^1, j = 0, 1$, construct $\mathcal{T}^2$ as follows:
(1) $Loc^2 = Loc^1 \cup \{l_{0,0}, l_{0,1}\}$,
(2) $Init^2 = Init^1$, (3) $V^2 = V^1$,
(4) $Trans^2 = (Trans^1 - \{\tau_0\}) \cup T^2$ with

$$T^2 = \{(l_0, g, aux_0 := x, l_{0,0}), \\ (l_{0,0}, \texttt{true}, aux_0 := aux_0 + 1, l_{0,1})\} \\ (l_{0,1}, \texttt{true}, x := aux_0, l_1)\}$$

Then $\mathcal{T}^1 \sim \mathcal{T}^2$. □

The next theorem indicates how unconditional assignment-free jumps between processing locations can be eliminated.

Theorem 2 *Let $l_0, l_1, l_2 \in Loc_P^1$ and $\tau_0 = (l_0, g, a, l_1), \tau_1 = (l_1, \texttt{true}, \varepsilon, l_2) \in Trans^1$, where ε denotes the empty assignment ("NOOP"). Assume that τ_i are the only transitions emanating from $l_i, i = 0, 1$. Then construct $\mathcal{T}^2$ as follows:*
(1) $Loc^2 = Loc^1 - \{l_1\}$,
(2) $Init^2 = Init^1$, (3) $V^2 = V^1$,
(4) $Trans^2 = (Trans^1 - \{\tau_0, \tau_1\}) \cup T^2$ with $T^2 = \{(l_0, g, a, l_2)\}$.
Then $\mathcal{T}^1 \sim \mathcal{T}^2$. □

The following theorem will be used in the next section to show that transitions with guard conditions of type $x < c$ with constant values c can be equivalently modelled using zero flag and sign flag as in compare/jump instruction pairs used on assembler level.

Theorem 3 *Let $\tau_0 = (l_0, x \leq c, a, l_1), \tau_1 = (l_0, x > c, b, l_2) \in Trans^1$ with constant value c. Assume that on all $\mathcal{T}^1$-paths emanating from l_1, l_2 variables $aux_0, ZF, SF \in V_L^1$ are first written to in an assignment before being used in a guard or a right-hand side assignment term. Then, using fresh location symbols $l_{0,j} \notin Loc^1, j = 0, 1$, construct $\mathcal{T}^2$ as follows:*
(1) $Loc^2 = Loc^1 \cup \{l_{0,0}, l_{0,1}\}$,
(2) $Init^2 = Init^1$, (3) $V^2 = V^1$,
(4) $Trans^2 = (Trans^1 - \{\tau_0, \tau_1\}) \cup T^2$ with

$$\begin{aligned}T^2 = \{&(l_0, \mathtt{true}, aux_0 := x, l_{0,0}),\\ &(l_{0,0}, \mathtt{true},\\ &\quad (ZF, SF) := (aux_0 = c, aux_0 < c),\\ &\quad l_{0,1}),\\ &(l_{0,1}, ZF \vee SF, a, l_1),\\ &(l_{0,1}, \neg(ZF \vee SF), b, l_2)\}\end{aligned}$$

Then $\mathcal{T}^1 \sim \mathcal{T}^2$. □

The following theorem is used to prove that indexed loops of type `for(i=0;i<c;i++)S;` can be equivalently modelled by compares and jumps as used on assembler level.

Theorem 4 *Suppose that S is a region of $\mathcal{T}^1$ with locations and associated transitions, so that the only transition entering S is $\tau_0 = (l_0, i \leq c, \varepsilon, l_1) \in Trans^1$ (c a constant) and the only transition leaving S is $\tau_1 = (l_2, \mathtt{true}, i := i + 1, l_0) \in Trans^1$. Assume further that $\tau_2 = (l_0, i > c, \varepsilon, l_3)$ and $\tau_3 = (l_4, \mathtt{true}, i := 0, l_0)$ are in $Trans^1$. Finally assume that in S and on all post-states of l_3 variables $aux_0, ZF, SF \in V_L^1$ are first written to in an assignment before being used in a guard or a right-hand side assignment term.*

Then, using fresh location symbols $l_{0,j} \notin Loc^1, 0 \leq j \leq 4$ construct $\mathcal{T}^2$ as follows:
(1) $Loc^2 = Loc^1 \cup \{l_{0,j} \mid 0 \leq j \leq 4\}$,
(2) $Init^2 = Init^1$, (3) $V^2 = V^1$,
(4) $Trans^2 = (Trans^1 - \{\tau_0, \tau_1, \tau_2, \tau_3\}) \cup T^2$ with

$$\begin{aligned}T^2 = \{&(l_4, \mathtt{true}, i := 0, l_{0,0}),\\ &(l_{0,0}, \mathtt{true}, \varepsilon, l_0),\\ &(l_0, \mathtt{true}, aux_0 := i, l_{0,1}),\\ &(l_{0,1},\\ &\quad (ZF, SF) := (aux_0 = c, aux_0 < c),\\ &\quad l_{0,2}),\\ &(l_{0,2}, ZF \vee SF, \varepsilon, l_1),\\ &(l_{0,2}, \neg(ZV \vee SF), \varepsilon, l_3),\\ &(l_2, \mathtt{true}, aux_0 := i, l_{0,3}),\\ &(l_{0,3}, \mathtt{true}, aux_0 := aux_0 + 1, l_{0,4}),\\ &(l_{0,4}, \mathtt{true}, i := aux_0, l_0)\}\end{aligned}$$

Then $\mathcal{T}^1 \sim \mathcal{T}^2$.

Proof. The theorem follows directly from application of theorems 2, 3 and Corollary 2. The effect of these applications is shown in Fig. 2. □

5. OBJECT CODE VERIFICATION

In this section we will describe our method for object code verification. First we give an overview of the method and then we give the details.

5.1 Overview

Each time the code generator g is applied to a SystemC model $\mathcal{M}$ resulting in an assembler program $\mathcal{A} = g(\mathcal{M})$, we must prove that $\mathcal{A} = g(\mathcal{M})$ is a correct implementation of $\mathcal{M}$.

To define what *correct implementation* means, first, in Subsections 5.2 and 5.3, we explain how one can lift $\mathcal{M}$ and $\mathcal{A} = g(\mathcal{M})$ to IOTS models $\mathcal{T}(\mathcal{M})$ and $\mathcal{T}(\mathcal{A})$ using semantic rules, and in Subsection 5.4 we explain how a mapping $\alpha_{IO}^{\mathcal{M}}$ between I/O variables of $\mathcal{T}(\mathcal{A})$ and $\mathcal{T}(\mathcal{M})$ can be induced.

Having this in hand, we can give the definition:

Definition 1 *An assembler program $\mathcal{A} = g(\mathcal{M})$ is a* correct implementation *of a SystemC model $\mathcal{M}$ if $\mathcal{T}(\mathcal{A}) \sim_{\alpha_{IO}^{\mathcal{M}}} \mathcal{T}(\mathcal{M})$*

In Subsection 5.5 our strategy to prove such an equivalence is explained.

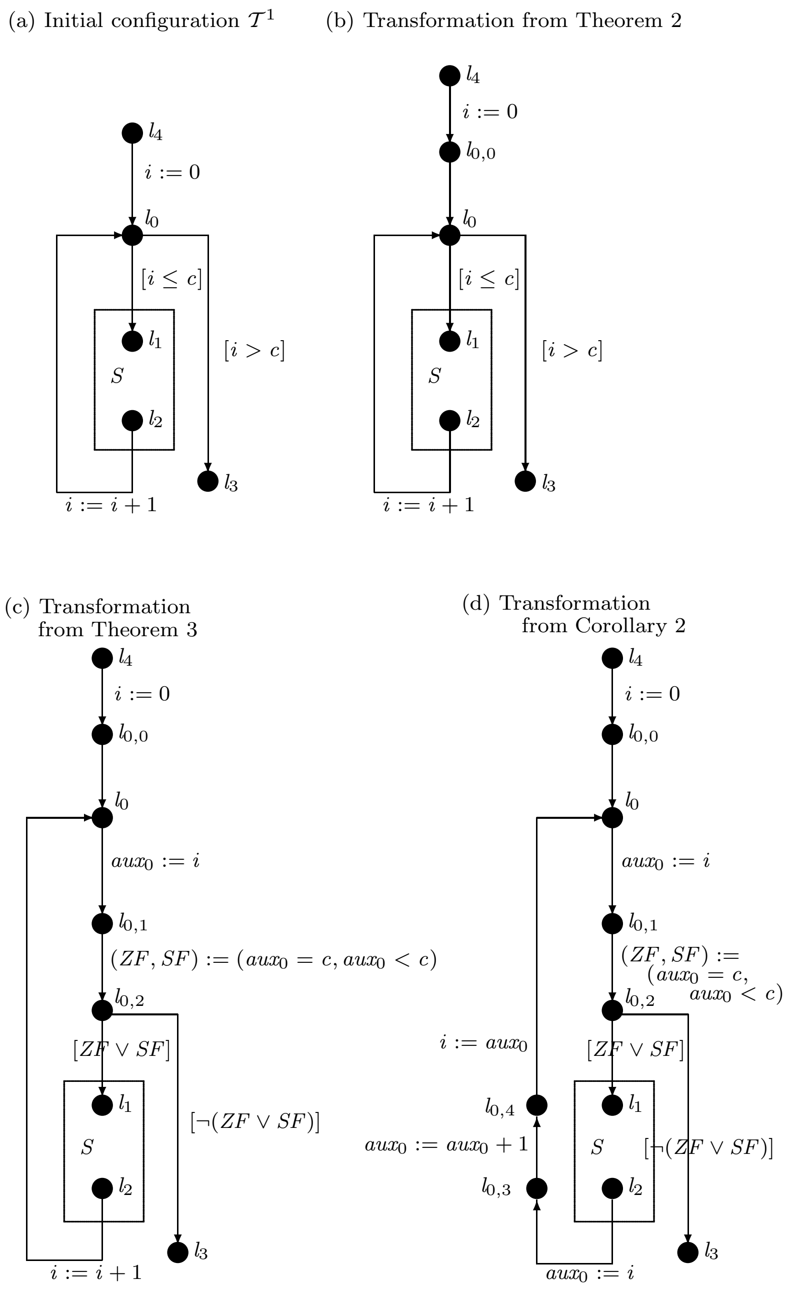


Fig. 2. I/O-equivalence preserving transformations in Theorem 4.

5.2 Semantics of SystemC models

While the semantic of general SystemC models is far more complex (Müller *et al.*, 2003), the behavioural semantics of SystemC programs observing the programming model above can be quite easily described using the I/O-safe transition systems introduced in Section 4 as behavioural model.

Hence, the behaviour of a generated SystemC program $\mathcal{M}$ observing the programming model described above can be represented as an IOTS model $\mathcal{T}(\mathcal{M}) = (\mathit{Loc}^{\mathcal{M}}, \mathit{Init}^{\mathcal{M}}, V^{\mathcal{M}}, \mathit{Trans}^{\mathcal{M}})$.

The set of variables $V^{\mathcal{M}} = V_I^{\mathcal{M}} \cup V_O^{\mathcal{M}} \cup V_P^{\mathcal{M}}$ with $V_P^{\mathcal{M}} = V_G^{\mathcal{M}} \cup V_C^{\mathcal{M}} \cup V_L^{\mathcal{M}}$ contains the following variable symbols: $V_I^{\mathcal{M}}$ contains the signals the controller uses to read the state of hardware devices, e.g. `actpt[n]` (actual state of point number `n`). $V_O^{\mathcal{M}}$ contains signals the controller uses to send requests to hardware devices, e.g. `reqpt[n]` (for sending requested state to point number `n`). $V_G^{\mathcal{M}}$ contains a shadow variable for each input and output variable, e.g. `actptNext[n]` and `reqptNext[n]` and defines internal state, such as array `res[n]` that keeps track of the reservation status of route number n. $V_C^{\mathcal{M}}$ contains some constants like conflict table entries `conflict[n]` used internally in the processing phase. $V_L^{\mathcal{M}}$ contains some local variables `i, r, ...` that are used as loop counters and temporary variables.

The set of locations $\mathit{Loc}^{\mathcal{M}}$ for program $\mathcal{M}$ is constructed by associating a label with *every* SystemC statement. The set of transitions $\mathit{Trans}^{\mathcal{M}}$ is now constructed by providing IOTS interpretations for each statement. (1) An assignment `l:x = e;` labelled l and followed by a statement labelled with l' leads to transition $(l, \texttt{true}, x := e, l')$. (2) A jump statement `l:goto L;` leads to the transition $(l, \texttt{true}, \varepsilon, L)$. (3) A guarded statement `l:if(b){ls:s;}` followed by a statement labelled with l' leads to the transitions associated with `ls:s` as well as the following transitions: (l, b, ε, ls) and $(l, \neg b, \varepsilon, l')$. Moreover, all exit transitions of `ls:s` are connected to l'. (4) A for statement `l:for (i=0; i < c ;i++){ls:s;}` followed by a statement labelled with l' leads to the transitions associated with `s` as well as the following transitions: $(l, \texttt{true}, i := 0, lc)$, $(lc, i >= c, \varepsilon, l')$, $(lc, i < c, \varepsilon, ls)$, $(li, \texttt{true}, i := i+1, lc)$, where lc and li are some labels not used elsewhere, and all transitions leaving `s` are connected to target location li.

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

5.3 Semantics of assembler programs

Any assembler program $\mathcal{A}$ generated by applying a conventional C/C++ compiler to a SystemC controller model $\mathcal{M}$ (that has been generated from a domain-specific model $\mathcal{D}$) adheres to a set of simple "low-level patterns". (This follows from the fact that the controller model $\mathcal{M}$ adheres to a number of coding patterns and interface restrictions.) For instance, the $\mathcal{A}$ program consists of a control loop that corresponds to the main loop of the SystemC program $\mathcal{M}$.

Assembler programs $\mathcal{A}$ can also be given a semantics in terms of an IOTS $\mathcal{T}(\mathcal{A}) = (\mathit{Loc}^{\mathcal{A}}, \mathit{Init}^{\mathcal{A}}, V^{\mathcal{A}}, \mathit{Trans}^{\mathcal{A}})$: The set of variables $V^{\mathcal{A}} = V_I^{\mathcal{A}} \cup V_O^{\mathcal{A}} \cup V_P^{\mathcal{A}}$ contains the following variable symbols:

$$
\begin{aligned}
V_I^{\mathcal{A}} &= \{x(,n,4) \mid x[n] \in V_I^{\mathcal{M}}\} \\
V_O^{\mathcal{A}} &= \{x(,n,4) \mid x[n] \in V_O^{\mathcal{M}}\} \\
V_P^{\mathcal{A}} &= V_G^{\mathcal{A}} \cup V_C^{\mathcal{A}} \cup V_L^{\mathcal{A}} \\
V_G^{\mathcal{A}} &= \{x(,n,4) \mid x[n] \in V_G^{\mathcal{M}}\} \\
V_C^{\mathcal{A}} &= \{x(,n,4) \mid x[n] \in V_C^{\mathcal{M}}\} \\
V_L^{\mathcal{A}} &= V_L^{\mathcal{M}} \cup \\
&\quad \mathit{REGS} \cup \mathit{FLAGS} \cup \mathit{SADDR}
\end{aligned}
$$

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 $\texttt{actpt} + 4 \cdot n$. In analogy to the SystemC variable concept, the processing variables $V_P^{\mathcal{A}}$ of the assembler program are also structured into global state variables, globally accessible constants and local variables. The latter, however, contain assembler-specific symbols in addition to the normal locals $i, j, r, \ldots$ used for counters and for storage of intermediate values: Set REGS contains all symbols denoting registers, such as `%eax, %edx, ...`. Set FLAGS contains the symbols `ZF, SF, PF, ...` for zero flag, sign flag, parity flag and others. The set SADDR contains stack address symbols used for auxiliary variables, written as `-n(%ebp)` where the `%ebp` register contains the stack base pointer and `-n` the offset from the base[1]. Valuations $\sigma^{\mathcal{A}} : V^{\mathcal{A}} \rightarrow D$ return the contents of the 4-bytes memory cells associated with address symbols like `actpt(,n,4), -n(%ebp), i, j, ...` and registers. All flags evaluate to 0 or 1.

It is easy to see that there is a 1-1 relationship between variable symbols in $V^{\mathcal{A}}$ and $V^{\mathcal{M}}$, except for the local variables, where $V^{\mathcal{A}}$ has additional variable symbols. Let $\alpha_{IO}^{\mathcal{M}} : V_I^{\mathcal{A}} \cup V_O^{\mathcal{A}} \rightarrow V_I^{\mathcal{M}} \cup V_O^{\mathcal{M}}$ denote the bijection between I/O variables in $\mathcal{A}$ and $\mathcal{M}$: It maps variables of the form $\mathtt{x}(,\mathtt{n},4)$ to variables of the form `x[n]`.

The set of locations $Loc^{\mathcal{A}}$ for program $\mathcal{A}$ is constructed by associating a label with *every* instruction of the assembler code.

The set of transitions $\mathit{Trans}^{\mathcal{A}}$ is now constructed by providing IOTS interpretations for each instruction. (1) A `l:movl a, b`-instruction (move contents of `a` to `b`[2]) labelled by l and followed by an instruction labelled l' results in IOTS transition $(l, \mathtt{true}, b := a, l')$. (2) A `l:jmp Lx`-instruction (unconditional jump to label `Lx`) leads to $(l, \mathtt{true}, \varepsilon, Lx)$, where ε denotes the empty assignment which does not change any variable valuation. (3) The `l:cmpl a,b`-instruction (compare `a,b` and set zero flag if $\mathtt{a} = \mathtt{b}$ and sign flag if $\mathtt{a} > \mathtt{b}$) followed by an instruction labelled with l' leads to transition $(l, \mathtt{true}, (\mathtt{ZF}, \mathtt{SF}) := (a = b, a > b), l')$. (4) The `l:jle Lx`-instruction (conditional jump to `Lx`, jump if previous compare evaluated to "less or equal"), followed by an instruction labelled with l', gives rise to transitions $(l, \neg(ZF \vee SF), \varepsilon, l')$ and $(l, ZF \vee SF, \varepsilon, Lx)$. (5) The `l:incl i`-instruction (increment i by 1) followed by an instruction labelled with l', gives rise to transition $(l, \mathtt{true}, (i, ZF, SF) := (i+1, i = -1, i < -1), l')$[3]. 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`.)

5.4 Abstraction mappings

As previously noted there is a 1-1 correspondence between SystemC symbols in $V^{\mathcal{M}}$ and assembler symbols in $V^{\mathcal{A}}$, except that $V^{\mathcal{A}}$ contains additional local variables: flags, registers and stack addresses.

We now define a set of variable symbols $V^{\mathcal{M}^+}$ that extends $V^{\mathcal{M}}$ with local variable symbols corresponding to the additional flags, registers and stack addresses in $V^{\mathcal{A}}$:

$$\begin{aligned}
V^{\mathcal{M}^+} &= V_I^{\mathcal{M}^+} \cup V_O^{\mathcal{M}^+} \cup V_P^{\mathcal{M}^+} \\
V_I^{\mathcal{M}^+} &= V_I^{\mathcal{M}} \\
V_O^{\mathcal{M}^+} &= V_O^{\mathcal{M}} \\
V_P^{\mathcal{M}^+} &= V_G^{\mathcal{M}^+} \cup V_C^{\mathcal{M}^+} \cup V_L^{\mathcal{M}^+} \\
V_G^{\mathcal{M}^+} &= V_G^{\mathcal{M}} \\
V_C^{\mathcal{M}^+} &= V_C^{\mathcal{M}} \\
V_L^{\mathcal{M}^+} &= V_L^{\mathcal{M}} \cup \mathit{REGS} \cup \\
&\quad\ \mathit{CFLAGS} \cup \mathit{SADDR} \\
\mathit{CFLAGS} &= \{eax, \ldots\}
\end{aligned}$$

[1]Recall that the stack grows towards lower addresses.

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

[3]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 $\mathcal{A}$.

We then define a map $\alpha^{\mathcal{M}}$ from $V^{\mathcal{A}}$ to $V^{\mathcal{M}+}$: The array elements `x(,n,4)` in $V^{\mathcal{A}} - V_L^{\mathcal{A}}$ are mapped to SystemC array elements $\alpha^{\mathcal{M}}(\texttt{x(,n,4)}) = \texttt{x[n]}$. It is the identity for variable symbols in $V_L^{\mathcal{A}} - FLAGS$. Flags $\%n$ in $FLAGS$ are mapped to flags n, e.g. $\alpha^{\mathcal{M}}(\texttt{\%eax}) = \texttt{eax}$.

Clearly $\alpha^{\mathcal{M}}$ is a bijection that preserves the status of variables (input/outpout/processing), and its restriction to I/O variables is equal to $\alpha_{IO}^{\mathcal{M}}$ defined in Section 5.3: $\alpha^{\mathcal{M}} \mid V_I^{\mathcal{A}} \cup V_O^{\mathcal{A}} = \alpha_{IO}^{\mathcal{M}}$.

5.5 Equivalence proof

A mechanised proof is automatically performed, making use of the generic theory introduced in Section 4, that is, a collection of valid assertions which are universally quantified over specific sets of parameters. The theory has been manually proven once, as sketched in Section 4. SystemC controller model $\mathcal{M}$ is now 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 rules for SystemC and assembler statements, respectively. Next, the symbols of $\mathcal{T}(\mathcal{A})$ are changed to C-style notation according to mapping $\alpha^{\mathcal{M}}$ defined above – 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 mechanised proof generator analyses $\mathcal{T}^1$ with respect to applicable patterns for theorems and corollaries of the generic theory. Each pattern application results in a I/O-equivalent transformation $\mathcal{T}^1 \mapsto \mathcal{T}^2 \mapsto \dots$ until the last transformation results in $\mathcal{T}(\mathcal{M}^+)$, whereupon the proof generator terminates.

We illustrate this mechanised proof procedure using a fragment from SystemC controller code, where 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_SIGNALS=3` and `NUM_POINTS=3`. From that the compiler[4] 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 of $\mathcal{A}$ is constructed by using the semantic rules for assembler instructions listed above. After changing the names of assembler variables to C-style notation according to mapping $\alpha^{\mathcal{M}}$ defined above, this results in an IOTS $\mathcal{T}^1$ which is depicted in Fig. 3. (2) Applying Theorem 1 to the regions S_{11} and S_{12} of $\mathcal{T}^1$ results is an I/O-equivalent IOTS $\mathcal{T}^2$ depicted in Fig. 4. (3) Twofold application of Theorem 4 on $\mathcal{T}^2$ results in I/O-equivalent IOTS $\mathcal{T}^3$ shown on the left-hand side of Fig. 5. Finally, a

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

valuation-preserving change of guard conditions ($[i \leq 2] \mapsto [i < 3], [i > 2] \mapsto [i \geq 3]$ etc.) yields $\mathcal{T}(\mathcal{M})$ which completes the proof, as far as the code fragments shown here for illustration purposes are concerned.

6. CONCLUSION

In this paper we have described a method for automated object code verification for railway controllers. 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, the main idea was to map $\mathcal{A}$ and $\mathcal{M}$ to their behavioural models $\mathcal{T}(\mathcal{A})$ and $\mathcal{T}(\mathcal{M})$ given in terms of some common semantic foundations and then prove that $\mathcal{T}(\mathcal{A})$ and $\mathcal{T}(\mathcal{M})$ are equivalent by applying transformations that have been proven once and for all to preserve I/O behaviour.

As common semantic foundations for the considered SystemC models and assembler programs we used I/O-safe transition systems. This semantics of SystemC models is much simpler than the semantics of general SystemC models, and it was only possible to use it because the considered SystemC models adhere to a restrictive programming model which can be practically enforced since our SystemC "programs" are automatically generated from higher-level domain-specific models. Using I/O-safe transition systems has the advantage that several practically relevant equivalence transformations can be applied, and the equivalence proofs are far more easy than those required for general transition systems.

REFERENCES

Badban, Bahareh, Martin Fränzle, Jan Peleska and Tino Teige (2006). Test automation for hybrid systems. In: *Proceedings of the Third International Workshop on SOFTWARE QUALITY ASSURANCE (SOQUA 2006)*. Portland Oregon, USA.

Berkenkötter, Kirsten (2006). OCL-based validation of a railway domain profile. In: *OCLApps 2006 - OCL for (Meta-)Models in Multiple Application Domains*. Accepted for publication.

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

Goos, Gerhard and Wolf Zimmermann (1999). Verification of compilers. In: *Correct System Design*. pp. 201–230. Springer.

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

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

Müller, W., J. Ruf and W. Rosenstiel (2003). *SystemC – Methodologies and Applications*. Chap. 4, pp. 97–126. Kluwer Academic Publishers.

Peleska, Jan, Daniel Große, Anne E. Haxthausen and Rolf Drechsler (2004). Automated verification for train control systems. In: *Proceedings of the FORMS/FORMAT 2004 - Formal Methods for Automation and Safety in Railway and Automotive Systems* (E. Schnieder and G. Tarnai, Eds.). Technical University of Braunschweig. pp. 252–265. ISBN 3-9803363-8-7.

Pnueli, Amir, Ofer Shtrichman and Michael Siegel (1998). The code validation tool CVT: Automatic verification of a compilation process. *International Journal on Software Tools for Technology Transfer* **2**(2), 192–201.

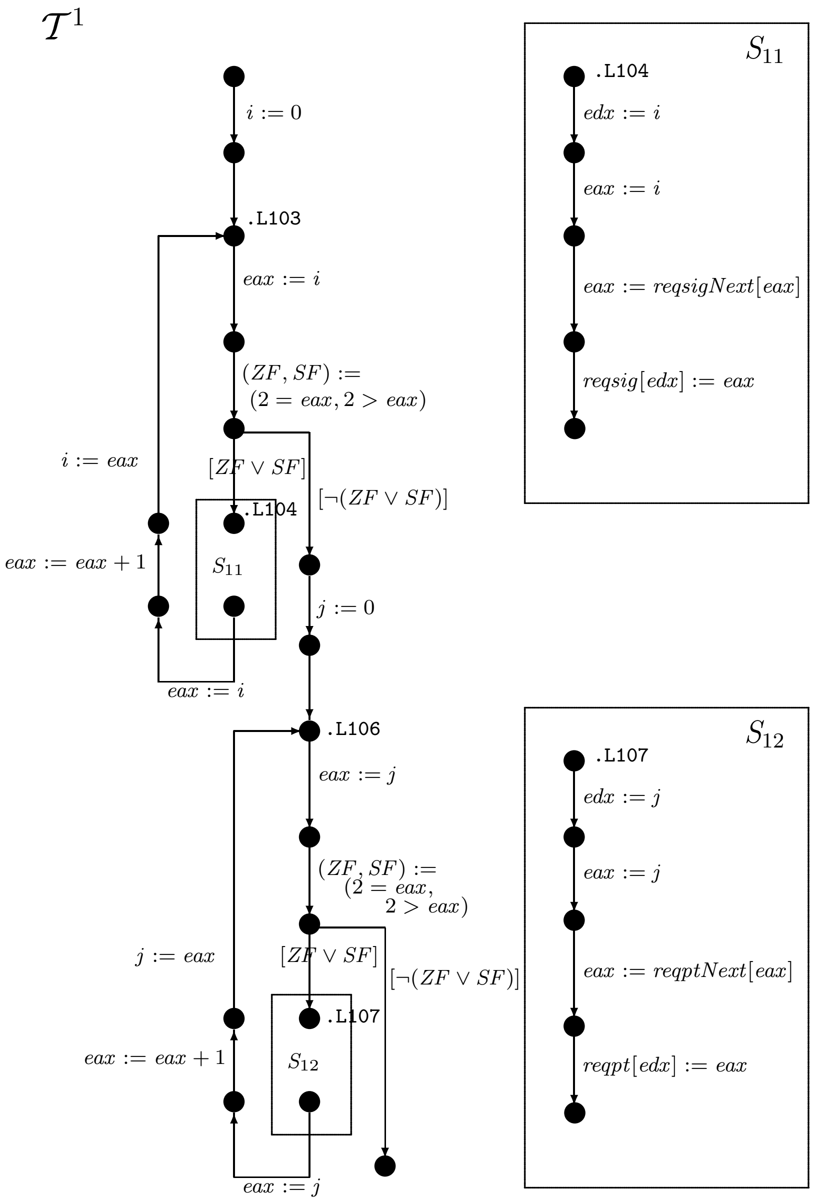
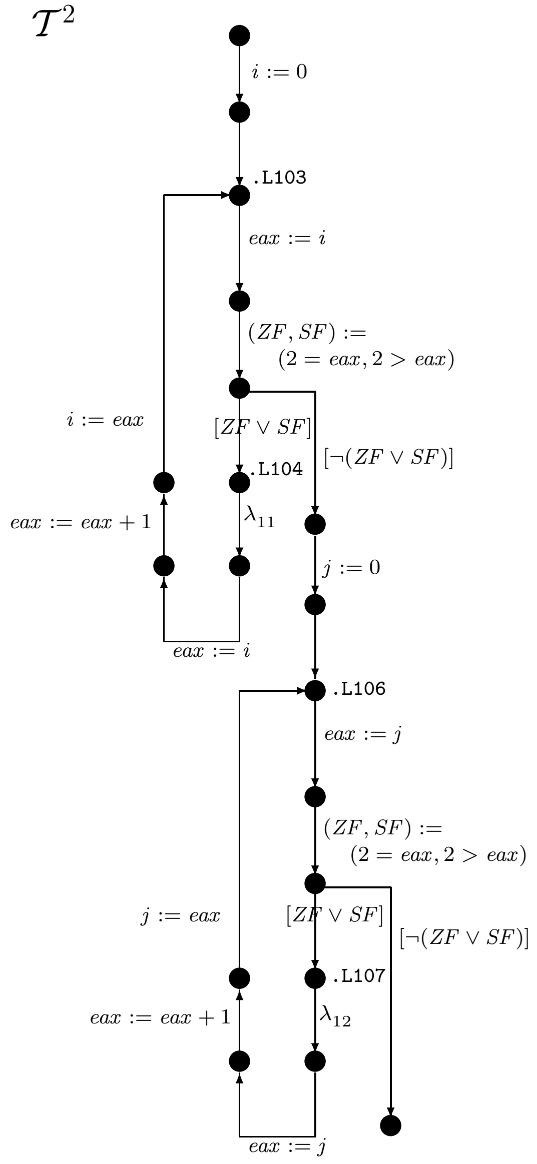
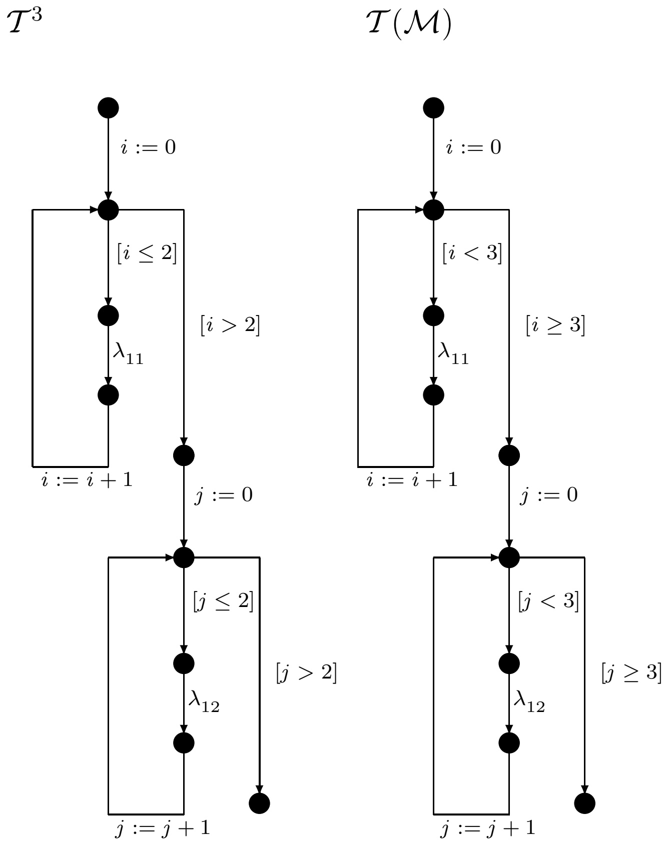


Fig. 3. IOTS $\mathcal{T}^1$ associated with $\mathcal{A}$ after renaming of variables.



$\lambda_{11} =_{def}$ *reqsig*$[i] :=$ *reqsigNext*$[i]$
$\lambda_{12} =_{def}$ *reqpt*$[j] :=$ *reqptNext*$[j]$

Fig. 4. $\mathcal{T}^1 \mapsto \mathcal{T}^2$: I/O-equivalent transformation according to Theorem 1.



$\lambda_{11} =_{def} \mathit{reqsig}[i] := \mathit{reqsigNext}[i]$
$\lambda_{12} =_{def} \mathit{reqpt}[j] := \mathit{reqptNext}[j]$

Fig. 5. $\mathcal{T}^3 \mapsto \mathcal{T}(\mathcal{M})$: I/O-equivalent guard transformation.