Modular Indirect Push-button Formal Verification of Multi-threaded Code Generators

Anton Wijs[0000−0002−2071−9624] and Maciej Wiłkowski

Eindhoven University of Technology, 5600 MB Eindhoven, The Netherlands
`a.j.wijs@tue.nl, m.wilkowski@student.tue.nl`

Abstract. In model-driven development, the automated generation of a multi-threaded program based on a model specifying the intended system behaviour is an important step. Verifying that such a generation step semantically preserves the specified functionality is hard. In related work, code generators have been formally verified using theorem provers, but this is very time-consuming work, should be done by an expert in formal verification, and is not easily adaptable to changes applied in the generator. In this paper, we propose, as an alternative, a push-button approach, combining equivalence checking and code verification with previous results we obtained on the verification of generic code constructs. To illustrate the approach, we consider our SLCO framework, which contains a multi-threaded Java code generator. Although the technique can still only be applied to verify individual applications of the generator, its push-button nature and efficiency in practice makes it very suitable for non-experts.

1 Introduction

Model-driven software development (MDSD) [23] aims to make the software development process more transparent and less error-prone. In an MDSD workflow, Domain Specific Languages (DSLs) are used to model the system under development, and model transformations are applied to initially refine the model, and finally generate source code that either fully or partially implements the program. The development of concurrent software is particularly complex, and techniques to support developers are sorely needed. Formal verification can play a vital role in that regard, to ensure that the artifacts produced in an MDSD workflow are functionally correct.

The correctness of models and source code has been investigated for many years, for instance see [8,19,21,25,46]. On the other hand. the *transformation* of a model to another model or code has received less attention [2]. To ensure that the final program is correct, it must be proven that the source code captures the intended functionality as specified by the models.

Verifying model tranformations, that transform an artifact into another artifact, is fundamentally more complex than verifying the artifacts themselves [2]. This is particularly true for model-to-code transformations (or code generators), due to the usual difference in abstraction level between input model and output

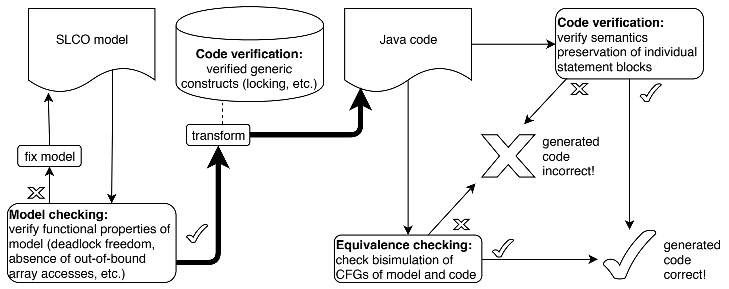

SLCO model
Code verification:
verified generic constructs (locking, etc.)
Java code
Code verification:
verify semantics preservation of individual statement blocks
transform
fix model
generated code incorrect!
Model checking:
verify functional properties of model (deadlock freedom, absence of out-of-bound array accesses, etc.)
Equivalence checking:
check bisimulation of CFGs of model and code
generated code correct!

Fig. 1. Model-to-code verification workflow.

source code, and the usual lack of formal semantics of the target programming language. Still, in recent years, techniques have been developed to *directly* verify code generators (and compilers) [7, 9, 24, 26, 27]. These techniques use theorem provers [6, 15, 30, 37]. Their advantage is that they can establish that the generators are guaranteed to provide correct output, but their drawback is the effort that is required to construct the proofs, the expertise needed to do so, and their inflexibility when the requirements of the generator change.

Alternatively, *indirect* approaches try to prove for a concrete input model that a generator produces correct output, every time the generator is applied. This is in practice often good enough, as the programs produced by generators are deployed, not the generators themselves, and it is much less complex to verify the output of a transformation rather than the transformation itself [2, 7]. However, most existing indirect approaches do not support multi-threaded code, have limited scalability, or check the preservation of particular properties, as opposed to full semantics preservation [1, 13, 33, 38–40].

In this paper, we propose an indirect technique to verify semantics preservation for generators of multi-threaded code. Its main features are that 1) it is push-button, requiring no additional input from the user when the generator is applied, and 2) it is modular, meaning that it scales linearly as the program size increases. We demonstrate our technique in the context of the *Simple Language of Communicating Objects* (SLCO) framework [35], which includes a generator for multi-threaded Java code, but the technique can be adapted to other DSLs and programming languages.

An overview of the technique workflow is given in the Activity diagram of Fig. 1. Initially, a given model is formally verified, by means of the mCRL2 model checker [11], to determine whether it satisfies a list of desired properties (for more information on this, see [35]). If it does, it can be subjected to code generation. Verification of this step is the topic of the current paper, and is done in two procedures that can be performed independently. In one procedure, for each state machine in the model, corresponding with one thread in the source code, a *control flow graph* (CFG) is extracted from both the model and the code. These CFGs are interpreted as Kripke structures, converted to Labelled Transition Systems, and finally compared by means of *bisimulation checking* [16, 31]. In the other procedure, implementations of individual state changes in the model are formally verified by means of the code verifier VerCors [8]. The separation logic specifications of those implementations, expressing the semantics of the model, are automatically generated. Together, the two verification results imply that the individual threads have been correctly implemented. Interaction between the threads is guaranteed to be correct, since the generator uses a mechanism for this that we have proven to be correct in the past, by means of the VeriFast code verifier [21,47]. All in all, we exploit the strengths of model checking, equivalence checking, and code verification to achieve verified MDSD.

Structure of the paper. Section 2 presents the preliminary concepts. Slco and code generation are discussed in Section 3. In this section, the formal semantics of Slco and an updated code generator are presented for the first time. Our code generator verification technique is explained in Section 4. Its implementation and experimental results are discussed in Section 5. Related work is considered in Section 6, and finally, Section 7 contains our conclusions.

2 Preliminaries

The semantics of a system can be formally expressed by a Labelled Transition System (LTS) as presented in Def. 1.

Definition 1 (Labelled Transition System). *An* LTS $\mathcal{G}$ *is a tuple* $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \hat{s})$*, with*

- $\mathcal{S}$ *a finite set of states;*
- $\mathcal{A}$ *a set of action labels;*
- $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ *a transition relation;*
- $\hat{s} \in \mathcal{S}$ *the initial state.*

Action labels in $\mathcal{A}$ are denoted by a, b, c, etc. A transition $(s, a, s') \in \mathcal{T}$, or $s \xrightarrow{a} s'$ for short, denotes that LTS $\mathcal{G}$ can move from state s to state s' by performing the a-action.

To compare LTSs, we use strong bisimulation, which is an equivalence relation, i.e., it is reflexive, symmetric and transitive.

Definition 2 (Strong bisimulation). *A binary relation B between two LTSs $\mathcal{G}_1 = (\mathcal{S}_1, \mathcal{A}_1, \mathcal{T}_1, \hat{s}_1)$ and $\mathcal{G}_2 = (\mathcal{S}_2, \mathcal{A}_2, \mathcal{T}_2, \hat{s}_2)$ is a* strong bisimulation *iff for all $s \in \mathcal{S}_1$ and $t \in \mathcal{S}_2$, $s\ B\ t$ implies:*

1. *if $s \xrightarrow{a} s'$ then $t \xrightarrow{a} t'$ and $s'\ B\ t'$;*
2. *if $t \xrightarrow{a} t'$ then $s \xrightarrow{a} s'$ and $s'\ B\ t'$.*

Two states s, t are (strongly) bisimilar, denoted by $s \leftrightarroweq t$, iff there is a bisimulation relation B such that $s\ B\ t$. Two LTSs $\mathcal{G}_1 = (\mathcal{S}_1, \mathcal{A}_1, \mathcal{T}_1, \hat{s}_1)$, $\mathcal{G}_2 = (\mathcal{S}_2, \mathcal{A}_2, \mathcal{T}_2, \hat{s}_2)$ are (strongly) bisimilar, denoted by $\mathcal{G}_1 \leftrightarroweq \mathcal{G}_2$, iff $\hat{s}_1 \leftrightarroweq \hat{s}_2$.

An alternative way to define the semantics of systems is by means of a *Kripke structure*, which is labelled on the states as opposed to the transitions.

Definition 3 (Kripke structure). *A* Kripke structure *is a tuple* $\mathcal{K} = (\mathcal{S}, \mathcal{P}, \mathcal{T}, \mathcal{L}, \hat{s})$*, with*

- $\mathcal{S}$ *a finite set of states;*
- $\mathcal{P}$ *a finite set of atomic propositions;*
- $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$ *a total transition relation;*
- $\mathcal{L} : \mathcal{S} \to 2^{\mathcal{P}}$ *a state labelling function.*
- $\hat{s} \in \mathcal{S}$ *the initial state.*

With $s \to t$, we denote that $(s, t) \in \mathcal{T}$. We refer to the domain of LTSs as **LTS**, and to the domain of Kripke structures as **KS**.

In [29, 36], a translation from Kripke structures to LTSs is defined that preserves bisimilarity.[1]

Definition 4 (Kripke structure into LTS embedding). *The* embedding $\boldsymbol{lts} : \mathbf{KS} \to \mathbf{LTS}$ *is defined as* $\boldsymbol{lts}(K) = (\mathcal{S}', \mathcal{A}, \mathcal{T}, \hat{s})$ *for arbitrary Kripke structures* $K = (\mathcal{S}, \mathcal{P}, \mathcal{T}, \mathcal{L}, \hat{s})$*, where*

- $\mathcal{S}' = \mathcal{S} \cup \{\bar{s} \mid s \in \mathcal{S}\}$*, with for all* $s \in \mathcal{S}$*, we have* $\bar{s} \notin \mathcal{S}$*;*
- $\mathcal{A} = 2^{\mathcal{P}} \cup \{\perp\}$*;*
- $\mathcal{T} \subseteq \mathcal{S}' \times \mathcal{A} \times \mathcal{S}'$ *is the least relation satisfying the following rules for all* $s, t \in \mathcal{S}$*:*

$$\frac{}{s \xrightarrow{\perp} \bar{s}} \qquad \frac{}{\bar{s} \xrightarrow{\mathcal{L}(s)} s} \qquad \frac{s \to t}{s \xrightarrow{\mathcal{L}(t)} t}$$

The fresh symbol $\perp$ is used to indicate that from the target state of a $\perp$-transition, an outgoing transition will be present with the original (Kripke) label of the target state of the latter transition.

3 Slco and the Generation of Java Code

Fig. 2 presents the meta-model of version 2.0 of Slco. An Slco model consists of a number of *classes*, instances of those classes called *objects*, multiple *channels* via which these objects can communicate with each other, and user-defined *actions*. Each class specifies the potential behaviour of a system component, and consists of a finite number of *state machines*, a set of *object-local variables* that can be accessed by each state machine in the class, and *ports* that are connected to channels, via which state machines can communicate with state machines in other objects. Variables can be of type *Boolean*, *Integer* or *Byte*, or Array of any of those types.

A state machine consists of a number of *states*, including one *initial state*, and *transitions* between those states, indicating possible state changes. Furthermore, a state machine may have a finite number of *state machine-local variables*,

[1] We omit a definition of bisimilarity for Kripke structures. For the details, see [36]. Also, in contrast to [29,36], the translation as defined here does not treat transitions between equally labelled states as internal LTS steps, since no such transitions are present in our Kripke structures (see Section 4.2).

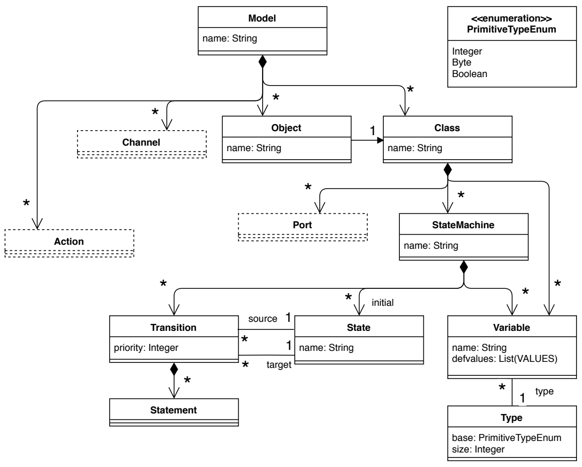


Fig. 2. The metamodel of SLCO 2.0.

and with each transition, a (possibly empty) sequence (or *block*) of *statements* is associated. Those statements can access (read and update) both the state machine-local variables and the variables of the instance (object) of the class in which the state machine resides. A transition can be *fired* if the current state of the state machine is the source state of the transition, and the associated block is *enabled*. A block is enabled iff its first statement is enabled. The order in which the outgoing transitions of a particular state should be considered for firing can be specified using *transition priorities*, but we do not consider these in the current paper. Here, we consider three types of statement:

– **Expression**: a statement evaluating to either **true** or **false**, e.g. $x = 0$. It is enabled iff it evaluates to **true**.
– **Assignment**: a statement assigning a value to either a state machine-local or object-local variable, e.g. $x := 1$. It is always enabled.
– **Composite**: a sequence of assignments, optionally preceded by an expression, e.g. $[x = 0; x := 1]$. It is enabled iff its first substatement is enabled.

In the current paper, we do not consider channels and ports (hence those concepts are marked with dashed lines in Fig. 2), focussing instead completely on concurrent behaviour represented by multiple state machines within a class. Hence, in the following, we do not discuss channels, nor do we consider actions. For more information on these concepts, see the SLCO tool paper [35]. The complete SLCO language also has the statements **send** and **receive**, for sending and receiving messages over channels. Nevertheless, in Section 4, we explain that including channels (and hence **send** and **receive** statements), actions and transition priorities actually requires only a minor extension of our verification approach. In other words, the fragment of SLCO that we focus on in this paper is sufficient to demonstrate our approach.

$$\text{assign}\,\frac{s \Rightarrow (x := e)s' \wedge \downarrow_\sigma (x := e)}{\langle \sigma, s\rangle \xrightarrow{x:=e} \langle \sigma[\xi_\sigma(e)/x], s'\rangle} \qquad \text{expr}\,\frac{s \Rightarrow (e)s' \wedge \xi_\sigma(e)}{\langle \sigma, s\rangle \xrightarrow{e} \langle \sigma, s'\rangle}$$

$$\text{comp}\,\frac{s \Rightarrow ([e; x_1 := e_1; \ldots; x_n := e_n])s' \wedge \xi_\sigma(e) \wedge \downarrow_\sigma (x_1 := e_1; \ldots; x_n := e_n)}{\langle \sigma, s\rangle \xrightarrow{[e;x_1:=e_1;\ldots;x_n:=e_n]} \langle \sigma[\xi_\sigma(e_1)/x_1]\cdots[\xi_\sigma(e_n)/x_n], s'\rangle}$$

$$\text{nonterm}\,\frac{s \Rightarrow (E)s' \wedge \uparrow_\sigma (E)}{\langle \sigma, s\rangle \xrightarrow{E} \lightning} \qquad \text{par}\,\frac{\langle \sigma, s, S\rangle \xrightarrow{E} \langle \sigma', s', S'\rangle}{\begin{array}{c}\langle \sigma, s||t, S, T\rangle \xrightarrow{E} \langle \sigma', s'||t, S', T\rangle \\ \langle \sigma, t||s, S, T\rangle \xrightarrow{E} \langle \sigma', t||s', S', T\rangle\end{array}}$$

Fig. 3. Slco SOS rules for assignment, expression, and composite statements

When a transition is fired, its block of statements is executed. The formal semantics of this for *basic* Slco, a version of Slco in which each transition has up to one statement, is presented by means of SOS rules in Fig. 3. Slco models can be transformed to basic Slco models by introducing additional states and transitions, and breaking up multiple-statement transitions into single-statement transition sequences. For the SOS rules, we denote with $s \Rightarrow (E)s'$ that in the state machine, there is a transition with statement E from state s to state s'. Furthermore, we reason about the current state of an Slco model by means of *situations*. A situation is a tuple $\langle \sigma, s_1|| \ldots ||s_n\rangle$, with σ a total function mapping the variables in the model to values of the appropriate types, and $s_1|| \ldots ||s_n$ the combination of the current states $s_1, \ldots, s_n$ of state machines 1 to n. The conclusions of the SOS rules are transitions in an LTS capturing the semantics of an Slco model, where the LTS states represent situations of the model.

The predicate $\downarrow_\sigma (E)$ evaluates to **true** iff execution of statement E under σ (i.e., the variables have the values defined by σ) successfully terminates. In particular, no out-of-bound accesses of array variables occur. The negation of this is denoted by $\uparrow_\sigma (E)$ and whenever this is applicable, trying to execute E results in reaching the *error situation* $\lightning$ (rule **nonterm**). Function $\xi_\sigma(e)$ is used to evaluate expression e, and in case e is of type Boolean, $\xi_\sigma(e)$ holds iff $\downarrow_\sigma (e)$ and e evaluates to **true**, and $\sigma[\xi_\sigma(e)/x]$ denotes an updated σ, in which $\xi_\sigma(e)$ has been assigned to variable x, the latter not being of type Array (but possibly an element of an array).

As indicated by SOS rule **par**, Slco has an interleaving semantics. If, in a given situation with a state s, a statement E can be fired, then so can it be fired in a situation consisting of the parallel composition of several states including s. Furthermore, note that the rules define that the execution of individual statements is *atomic*, i.e., cannot be interrupted by the execution of other statements.

Generation of multi-threaded Java code. The Slco framework includes a generator for multi-threaded Java code, in which each state machine in a given Slco model is transformed into a separate thread. Fig. 4 presents part of an example Slco model, named RE, on the left, and part of the translation of state machine SM1, contained in RE, on the right. Checking for the code-equivalent of enabled transitions and executing associated statement translations is done in the method exec as part of the SM1 Java thread. Depending on the current

```
1   model RE {
2     classes
3     P {
4       variables Boolean[3] x
5       state machines
6         SM1 {
7           variables Integer i
8           initial S0 states S1
9           transitions
10            S0 -> S1 {
11              [not x[i]; x[i+1]:=x[i]]
12            }
13            S0 -> S1 {}
14            S1 -> S0 {
15              i := (i+1) % 2;
16            }
17        }
18        SM2 { ... }
19    }
20    objects p: P(x:=[False,True,True])
21  }
```

```
1     ...
2   public void exec() {
3     // variable to store non-deterministic choices
4     int j_choice;
5     while(true) {
6       switch(j_currentState) {
7         case S0:
8           j_choice = j_randomGenerator.nextInt(2);
9           switch(j_choice) {
10            case 0:
11              if (execute_S0_0()) {
12                j_currentState = RE.j_State.S1;
13              }
14              break;
15            case 1:
16              if (execute_S0_1()) {
17                j_currentState = RE.j_State.S1;
18              }
19              break;
20          }
21        case S1:
22          if (execute_S1_0()) {
23            j_currentState = RE.j_State.S0;
24          }
25          break;
26        default:
27          return;
28      }
29    }
30  }
31    ...
```

Fig. 4. An example SLCO model (left) and part of its Java implementation (right)

state (j_currentState), the execution of the statements of a translated outgoing transition is attempted, and if successful, the associated state change is applied. Non-determinism is translated by the code generator by using a random number generator (line 8 in the code) to randomly select the code of an outgoing transition. Note that for each SLCO transition, a dedicated *transition method* is implemented that executes a translation of the block associated to the transition.

In Fig. 5, implementations of the transition methods execute_S0_0 and execute_S1_0 are given, which map one-to-one on the blocks in RE. Each method returns a Boolean value reflecting whether or not the statement block was successfully executed. Note that for shared (object-local) variables, a locking mechanism is required, to ensure that no inconsistent behaviour occurs due to multiple threads accessing and updating the same variables simultaneously. This locking mechanism is based on the concept of *ordered locking* [18]: each variable is associated with a separate lock, the locks are sorted, and acquiring locks should be done in that specified order. In the example, each element of array x has its own lock, and the locks for x[i] and x[i+1] both need to be acquired before the update can be performed. After adding the lock IDs to an array java_lockIDs (lines 3-4 in method execute_S0_0), and sorting the IDs (line 5), the locks are requested from the Java object java_kp (line 6). After evaluation and/or execution of the statement, the locks are released (lines 8 and 12).

Finally, instances of the java_Keeper class, such as java_kp in Fig. 5, manage the locks of the implementation of an SLCO object in an array of reentrant locks; given a number of lock IDs in an array l, a lock method tries to acquire the locks in the specified order, and an unlock method releases the locks in that order.

```
boolean execute_S0_0() {
  // [not x[i]; x[i + 1] := x[i]]
  java_lockIDs[0] = 0 + i;
  java_lockIDs[1] = 0 + i + 1;
  Arrays.sort(java_lockIDs,0,2);
  java_kp.lock(java_lockIDs, 2);
  if (!(!(x[i]))) {
    java_kp.unlock(java_lockIDs, 2);
    return false;
  }
  x[i + 1] = x[i];
  java_kp.unlock(java_lockIDs, 2);
  return true;
}
```

```
boolean execute_S1_0() {
  // i := (i + 1) % 2
  i = (i + 1) % 2;
  return true;
}
```

Fig. 5. Methods execute_S0_0 and execute_S1_0 for RE

4 Verification of Code Generation

4.1 Verification Overview

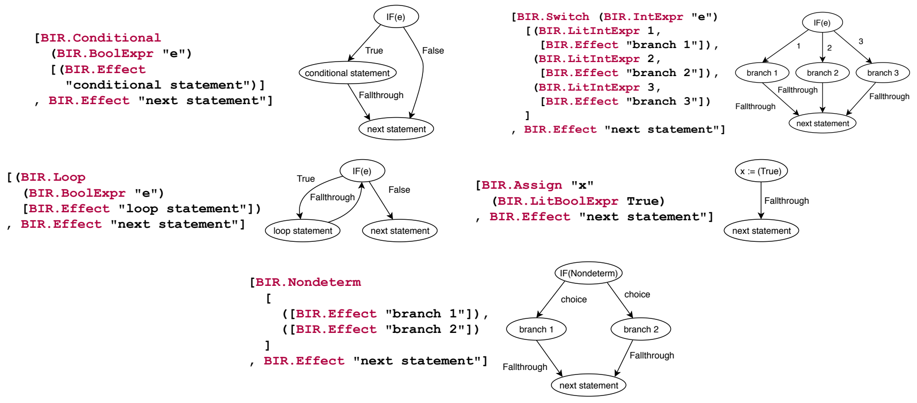
Fig. 1 presents an Activity diagram of our workflow for the verification of the code generator. In the current section, we discuss the various steps from the *transform* activity onwards, and reason about the fact that together, these steps provide a correctness proof for individual applications of the code generator. As input, we expect an Slco model that is functionally correct, i.e., that has the desired functional properties and the absence of out-of-bound array accesses.

The code generator uses a library of *generic* constructs that can be reused each time the generator is applied on a model. In general, the content of such a library is DSL-specific. For Slco, we have added implementations for the *channel* construct, and the ordered locking scheme. Implementations of generic constructs can be formally verified once, and then safely reused in each application of the generator. In the past, we have verified both constructs using the VeriFast code verifier [21,47]. For the ordered locking scheme, we have verified that no deadlocks can be introduced by locking, and that the scheme ensures that atomicity of the statements is preserved. Hence, as long as the generated code adheres to the scheme it is guaranteed that concurrent executions of statements do not interfere with each other, and that no deadlocks occur when trying to acquire locks. It is straightforward to check this for our generated code, since the locks are only accessed via generic `lock` and `unlock` methods that are part of the verified ordered locking scheme implementation, and that have been proven to implement lock acquisition and release correctly, using VeriFast.

Isolating the locking scheme has multiple advantages for the remaining verification task. First of all, the locking steps can be ignored, and we can focus on the behavioural aspects specified by the model. Second of all, it makes the remaining verification task modular; as the shared variables are the only means for the state machines in the model to communicate, the interaction of the corresponding threads in the program is guaranteed to be correct. Furthermore, as statement atomicity is preserved, we know that no inconsistent system states can be reached when the threads execute in parallel. What remains is to prove that each individual state machine is correctly translated into a thread.

```
data Ins = Conditional Expr Block
         | Switch Expr [(Expr, Block)]
         | Nondeterm [Block]
         | Loop Expr Block
         | Assign VariableRef Expr
         | MethodInv MethodCall
```

Fig. 6. Definition of the BIR main concepts



```
[BIR.Conditional
  (BIR.BoolExpr "e")
  [(BIR.Effect
    "conditional statement")]
, BIR.Effect "next statement"]
```

```
[BIR.Switch (BIR.IntExpr "e")
  [(BIR.LitIntExpr 1,
    [BIR.Effect "branch 1"]),
   (BIR.LitIntExpr 2,
    [BIR.Effect "branch 2"]),
   (BIR.LitIntExpr 3,
    [BIR.Effect "branch 3"])
  ]
, BIR.Effect "next statement"]
```

```
[(BIR.Loop
  (BIR.BoolExpr "e")
  [BIR.Effect "loop statement"])
, BIR.Effect "next statement"]
```

```
[BIR.Assign "x"
  (BIR.LitBoolExpr True)
, BIR.Effect "next statement"]
```

```
[BIR.Nondeterm
  [
    ([BIR.Effect "branch 1"]),
    ([BIR.Effect "branch 2"])
  ]
, BIR.Effect "next statement"]
```

Fig. 7. Translating BIR to CFG

To do the latter, each time the generator is applied on a given SLCO model, two verification steps can be performed independently: checking for equivalence of the CFGs of each state machine and its corresponding Java thread, to determine whether they have equivalent control flows, and verifying semantics preservation when the blocks of individual transitions are translated to Java methods, to determine whether the individual steps in the control flows are equivalent. In Section 4.2, we discuss the former. The latter is addressed in Section 4.3.

The two steps nicely complement each other, together addressing the semantics (Fig. 3). The CFG equivalence checking step establishes that in all reachable program states, each transition method will be considered for execution iff the corresponding transition is an outgoing transition of the corresponding model state. The transition method verification step establishes that execution of a transition method indeed has the intended effect on the current state, i.e., all transition methods correctly implement the block of their transition. In Section 4.4, we discuss what is required to support the complete SLCO language, including the use of channels, actions, and transition priorities.

4.2 Constructing and Comparing CFGs

To extract accurate CFGs from SLCO state machines and Java threads, we have defined the *Behaviour Intermediate Representation* (BIR) language. This language has enough concepts to capture both types of CFGs: on the one hand, it can reflect how the Java code implements the statements of an SLCO model,

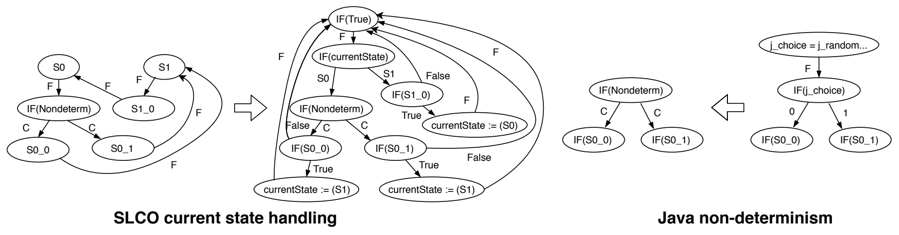


Fig. 8. Transforming the Slco and Java CFGs of RE.

and therefore support if-then, switch, and loop constructs. On the other hand, it supports modelling concepts such as non-deterministic choice.

Fig. 6 lists a definition of the main BIR instructions. An `Expr` can be evaluated and a `Block` is a sequence of instructions. Furthermore:

– A `Conditional` guards a `Block` with an `Expr` condition. If the latter evaluates to **true**, the `block` can be executed. It can be used to represent `if-then` Java constructs.
– A `Loop` expresses that execution of the involved `Block` should be repeated as long as the involved `Expr` condition evaluates to **true**.
– A `Switch` branches to multiple instructions, where the branches represent the different possible outcomes of evaluating the first `Expr` instruction. It can be used to represent Java `switch` constructs.
– A `Nondeterm` branches to multiple instructions non-deterministically. It can represent non-deterministic choice in Slco models.
– An `Assign` expresses that the evaluation of the `Expr` instruction should be assigned to the given variable reference.
– A `MethodInv` represents a method invocation.

BIR descriptions of state machines and threads can be represented by CFGs. How partial CFGs are derived from the various BIR instructions is shown in Fig. 7. For the `Conditional`, `Switch`, `Loop`, `Assign` and `Nondeterm` instructions, direct translations are given (`MethodInv` is processed similar to `Assign`), with `BIR.Effect` being used as a placeholder for nested instructions that need to be translated recursively. In addition, there are various types of expression, to reflect their type and whether they are literals or more complex expressions. In the CFGs, nodes are labelled with BIR instructions, and edges are either labelled with `Fallthrough`, representing unconditional flow of execution, `choice`, representing an option for a non-deterministic choice, or some value, representing the result of an evaluation for a branching instruction (IF-node).

While the CFG of a Java program can be directly obtained by transforming all constructs to BIR instructions, the transformation of Slco models is more involved: first, each statement block is transformed, after which for each state machine state, an additional instruction is created, and those state instructions are connected with each other via the BIR representations of the blocks. On the left in Fig. 8, the CFG of `SM1` (Fig. 4) is given, where `F` is short for `Fallthrough`, `C` is short for `choice`, and `S0_0`, `S0_1` and `S1_0` represent the various transitions.

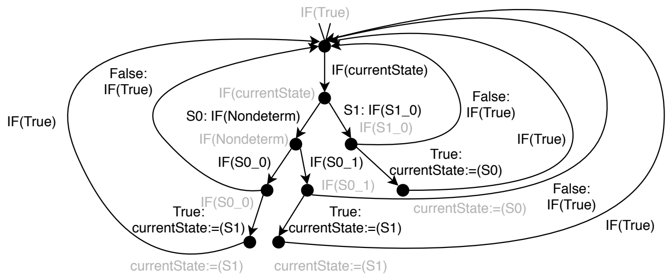


Fig. 9. The LTS produced for the RE example, both for the model and the code.

Once the CFGs of an SLCO model and corresponding generated Java code have been constructed, some transformations have to be applied on them to bring them semantically together. First of all, nodes representing SLCO blocks must have the same label as their corresponding Java translations (their actual semantic equivalence is addressed in Section 4.3). Besides this, we apply two other transformations: 1) we introduce a mechanism on the SLCO side to keep track of the current state, as this is also done on the Java side, and 2) we directly introduce nondeterminism on the Java side by means of the `Nondeterm` instruction. We have proven manually that both transformations are semantics preserving. Fig. 8 presents the application of those transformations on the RE example. On the SLCO side, the nodes representing state machine states are removed, and the control flow is replaced by a `switch` instruction involving a new variable `currentState`, to keep track of the current state, inside a `Loop` instruction with condition **true**, i.e., an infinite loop. The instructions representing the various blocks are turned into IF-nodes, and are connected to the new `switch` instruction via the appropriate value of `currentState`. After each block instruction, the current state is updated if execution of the block was successful. Otherwise, the state machine remains in the same state.

On the Java side, every occurrence of the `j_choice` variable, which involves obtaining a new random value followed by a `switch` instruction using `j_choice`, is replaced by a new `Nondeterm` instruction.

When the CFGs have been transformed, what remains is to remove the edge labelling, such that the CFGs can be interpreted as Kripke structures. Labels `Fallthrough` and `choice` can be safely removed, but the conditional labels need to be preserved, in order to maintain the guarded control flow. As the nested instructions inside `Conditional`, `Switch` and `Loop` all have exactly one incoming edge, coming from an IF-node (Fig. 7), we can move the conditional edge labels to the target nodes of those edges. For instance, in case of `Switch` in Fig. 7, we relabel the `branch 1` node to `1: branch 1`, and so on.

Finally, we transform the resulting Kripke structures to LTSs (Def. 4), and check whether the LTSs are strongly bisimilar (Def. 2). In case of the RE example, the two LTSs are in fact identical. Fig. 9 presents one of those LTSs. For ease of presentation, each state $\hat{s}$ (black circle) in Fig. 9 in fact represents a state

s together with a companion state $\bar{s}$, connected via a $\perp$-transition from s to $\bar{s}$ and a transition labelled with the associated grey label in Fig. 9 back from $\bar{s}$ to s. All in- and outgoing transitions of a state $\hat{s}$ in Fig. 9 are connected to s (as opposed to $\bar{s}$). The top state in the figure is the initial state. This is indicated by the large incoming arrow head.

4.3 Verification of Transition Methods

```
/*@
pure boolean updated_S0_0_x(int x_index, int i0, boolean b1,
  boolean x_old) {
        return ((x_index == i0) ? b1 : x_old);
}

given int i;
given boolean[] x;
given boolean b0;
given int i0;
given boolean b1;
invariant x != null;
context (\forall* int slco_i ; 0 <= slco_i < x.length ;
  Perm(x[slco_i],write));
ensures b0 ==> \result == true;
ensures !b0 ==> \result == false;
ensures b0 ==> (\forall* int slco_i ; 0 <= slco_i < x.length ;
  x[slco_i] == updated_S0_0_x(slco_i, i0, b1, \old(x[slco_i])));
ensures !b0 ==> (\forall* int slco_i ; 0 <= slco_i < x.length ;
  x[slco_i] == \old(x[slco_i]));
@*/
boolean execute_S0_0() {
  // SLCO statement: [not x[i]; x[i + 1] := x[i]]
  /*@ assume 0 <= i < x.length; @*/
  /*@ b0 = !(x[i]); @*/
  if (!(!(x[i]))) { return false; }
  /*@ assume 0 <= i + 1 < x.length; @*/
  /*@ assume 0 <= i < x.length; @*/
  /*@ i0 = i + 1; b1 = x[i]; @*/
  x[i + 1] = x[i];
  return true;
}

/*@
given int i;
given boolean[] x;
given int i_old;
invariant x != null;
ensures \result == true;
ensures (i == (i_old + 1) % 2);
@*/
boolean execute_S1_0() {
  // SLCO statement: i := (i + 1) % 2
  /*@ i_old = i; @*/
  i = (i + 1) % 2;
  return true;
}
```

Fig. 10. Methods `execute_S0_0` and `execute_S1_0` for RE, with VerCors specifications.

To verify the transformation of each Slco block, we use the VerCors tool set [8]. With it, we can check whether Java code satisfies a specification written in permission-based separation logic [4]. Its verification engine is Viper [28].

We have extended our Slco-to-Java code generator with a feature to generate a list of the transition methods implementing the Slco blocks of a given Slco model, with VerCors specifications. In Fig. 10, methods execute_S0_0 and execute_S1_0 of the RE example model are listed with their specifications. The specifications formally express the effect of executing Slco statements, as defined by the corresponding SOS rules (Fig. 3). Note the absence of locking, as this can be abstracted away safely, making it easier to construct VerCors specifications.

To isolate the methods from the complete program, we specify which variables each method can access (the given ... statements at lines 7-8 and 35-36). Furthermore, for all arrays, we specify that they have been properly initialised (invariant x != null, lines 12 and 38), and in case array elements are updated by the method, appropriate write permission is given (lines 13-14).

To properly express postconditions, ghost variables are used. In case of an Slco Assignment translation, such as execute_S1_0, the old value of the updated variable (here i) is stored (in a variable i_old), which allows us to specify the effect (line 40). Furthermore, we specify that **true** is returned (line 39).

In case a Composite statement is translated (execute_S0_0), the specification is more elaborate, as multiple variables can be updated, and there is optionally a guard. Ghost variables are used to store all intermediate results, such that they can be referred to in the specification. Note that before each array access, an assumption has been added to specify that no out-of-bound array accesses can be performed, relying on the Slco model having been verified in this regard. Depending on the evaluation of the guard, the method either returns **true** or **false** (lines 15-16), and the array is either updated or not (lines 17-20). Finally, when an array is updated in a Composite statement, an auxiliary function is defined (for example, see lines 2-5), since array elements may be updated multiple times in a single Composite, and in general, when expressions are used to compute array indices, this cannot be detected statically. In case an element is updated multiple times, only the final update should be specified in the post-condition, and the auxiliary function allows us to relate each element to their final update.

The final case, not applicable in the example, is that a method implements an Slco Expression. The postcondition of such a method addresses that **true** is returned iff a guard implementing the Expression evaluates to **true**.

4.4 Supporting the Complete Slco Language

The approach, as presented in the previous sections, verifies that a model written in a specific fragment of the Slco language is correctly transformed into multi-threaded Java code. To support the complete Slco language, the verification approach needs to be extended in a number of ways. In this section, we discuss these extensions, which require only minor changes to the approach as it exists currently. Implementing those extensions is planned for future work.

1. Slco has the concept of *channel* to model the communication between state machines of different objects by means of message passing. In earlier work,

we have formally verified that a (lock protected) Java channel correctly implements the semantics of the Slco channel [10]. This implementation is now part of our library of generic constructs. To support channels, the verification approach proposed in the current paper only needs to be extended to match **send** and **receive** statements in the CFGs of Slco state machines and their corresponding Java threads. As those statements are implemented using the verified `send` and `receive` methods of the generic implementation of Slco channel, the actual *effect* of sending and receiving a message via a channel does not need to be verified anymore.

2. User-defined *actions* are Slco statements that allow the definition of model-specific instructions. For instance, these can refer to calling standard Java library methods. Our verification approach can be extended straightforwardly to match the actions in the CFGs of the state machines and their corresponding Java threads.
3. Finally, Slco supports *transition priorities* that allow the user to specify the order in which transitions should be fired. In Java, this order is implemented by placing implementations of the associated statements inside `if-then-else` constructs. First of all, the BIR language (more specifically, the `Nondeterm` instruction) must be extended with priorities. Second of all, a transformation needs to be defined to transform `if-then-else` constructs implementing those priorities in Java to `Nondeterm` instructions, similar to how implementations of non-determinism are transformed to such instructions in the current approach.

5 Implementation and Experiments

The Slco framework has been developed in Python 3, using TextX [14] for meta-modelling and Jinja2[2] for model transformations. Hence, the generation of VerCors specifications has also been implemented in Python. The CFG extractor, including the transformations from CFG to Kripke structures and from Kripke structures to LTSs, has been written in Haskell. For bisimulation checking of LTSs, we use the mCRL2 toolset, which has a tool called *ltscompare* that implements efficient bisimulation checking with complexity $\mathcal{O}(m \log n)$, with n the number of states and m the number of transitions in an LTS [16, 31].

To validate the effectiveness of our approach, we ran a number of experiments on a MacBook Pro with a 3.1 GHz Intel Core i5 processor and 16 GB RAM, running macOS Mojave. We selected 50 models from the Beem benchmark suite [32]. These models stem from well-known examples and case studies, modelling mutual exclusion algorithms, communication protocols, controllers, leader election algorithms, planning and scheduling problems, and puzzles. Originally written in the DVE language, the models have first been translated to Slco using a model transformation. This is a straightforward task, as all language concepts of DVE can be translated to similar concepts in Slco. In most cases, a model contains a single object with one or more state machines. Furthermore, in many

[2] `http://jinja.pocoo.org`.

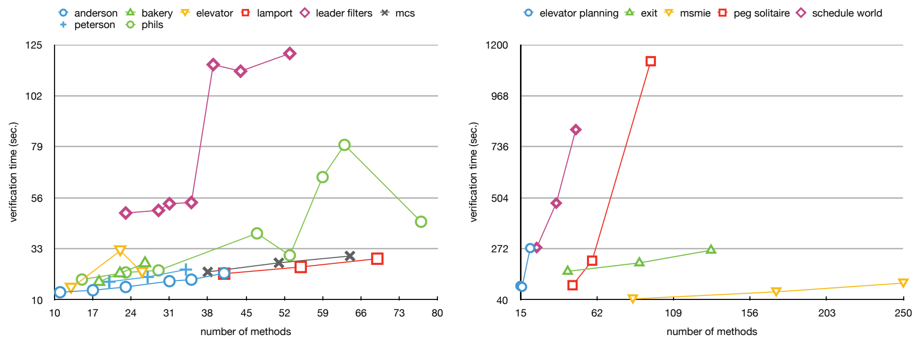


Fig. 11. VerCors verification runtimes

cases, thousands of lines of code were produced when translating the models to Java code. The largest instance overall that we considered, msmie.4, resulted in 12,157 lines of source code, implementing a program with ten threads.

Due to the modular approach of our verification procedure, allowing us to isolate individual state machines, the CFG equivalence checking step never required more than 0.5 seconds to process, given a model, all its state machines and their translations. Regarding the time required for the verification of transition methods, Fig. 11 presents the runtime results for VerCors for a representative selection of 13 models. For each model, we have processed multiple instances, between three and eight, that are all part of the Beem benchmark set. This allows us to investigate how the runtime scales as the number of transition methods is increased. For the other models, similar scalability results have been obtained. As expected, in most cases, the runtime scales linearly, but not to the same degree for all models. For instance, for the msmie instances, although they have many methods, the verification time is very short, since the methods are not complex, most of them containing only unguarded assignments. On the other hand, the peg solitaire instances have transition methods with guards and relatively complex expressions to refer to array elements, resulting in the runtime increasing much more rapidly as the number of methods is increased. For a few models, such as phils, this phenomenon results in the runtime not linearly increasing as the number of methods is increased. Some instances have more methods than others, yet fewer of those are guarded, or involve array accesses.

Concluding, the CFG equivalence checking step scales very well, and can be used to reason about the CFGs of large models and programs. Moreover, for the verification of transition methods, VerCors is very suitable, but to improve scalability, we will have to work on reducing the amount of verification work, for instance by detecting functional duplicates among the transition methods. In case of the Beem models, many models contain such duplicates. In those cases, the involved state machines are all very similar, specifying the same computation to be performed on different data. In future work, we plan to exploit that.

6 Related Work

Equivalence checking has been applied in the past to directly verify semantics preservation of model transformation rules, for instance see [5, 20, 34, 44]. This approach requires that a model transformation can be formally defined, and hence that both the source and target modelling language has a formal semantics. Furthermore, programming languages describe systems at a much lower abstraction level than modelling languages, making equivalence checking more directly suitable for model-to-model than for model-to-code transformations. In the current paper, we use equivalence checking as well, but we also apply CFG transformations and code verification to bridge the gap between abstraction levels.

For an overview of applying formal verification on model-to-code transformations, see [2]. Formal verification of a statechart-to-Java generation algorithm using the Isabelle/HOL theorem prover [30] is described in [7]. Similar to our approach, their proof aims to demonstrate bisimulation between model and code, but their modelling language does not support variables. They prove once that the generator algorithm is correct, but note that for full verification, it may be more suitable to verify on a case-by-case basis, like we do, to ensure that the implementation of the generator algorithm is correct as well.

In [39], a Java code generator framework based on QVT is presented. The KIV theorem prover [15] is used to prove particular security properties. Staats and Heimdahl [38], on the other hand, apply model checking on both the model and the code to verify the preservation of selected LTL properties. In [40], DSL-generated C code is checked using the Spin model checker [19]. A Promela model is generated in which the generated C code is embedded, and LTL properties are formulated for checking. Pnueli et al. [33] propose the CVT tool that uses refinement checking to detect whether properties, proven to be satisfied by a given Statemate model, have been preserved in generated C code. In [13], the preservation of properties is verified by means of transforming Event-B models [3] to specifications for the Dafny code verification tool [25].

In contrast to all the approaches above, we check for the preservation of the complete model semantics. In our view, a list of concrete properties can serve to verify that a model is correct, although the question always remains whether such a list completely covers the intended functionality. For a code generation step, on the other hand, it must be guaranteed that the generated code exactly implements what the model specifies. If this is the case, then any property satisfied by the model will be preserved by the code generator. Additionally, most of the above approaches do not support the generation of multi-threaded code, even though constructing such programs is particularly error-prone.

Ab Rahim and Whittle [1] propose an interactive technique in which the user initially supplies assertions about generated code that are later added automatically to code using a separate model tranformation. A software model checker is applied to verify that assertions hold in generated code. Our technique, on the other hand, does not require additional user input, and due to its modular approach, scales much better than one directly using model checking.

Techniques for compiler verification are similar to code generator verification techniques, and can be used to further strengthen the development workflow to verify that code is correctly compiled into an executable. In [26, 27], the Coq theorem prover [6] is used to both implement and verify a C compiler. A compiler for the block diagram language Lustre [17] is verified using Coq in [9]. Finally, Kumar et al. [24] use the HOL4 theorem prover [37] to verify that programs described in their language, called CakeML, are compiled correctly. These results are impressive, yet we question how flexible the techniques are when the code generation or compilation procedure needs to be updated.

Finally, software model checking techniques, such as [12, 22], offer another approach to verify code. Tools such as Java PathFinder [41] could be useful to verify parts of the generated code. We plan to investigate how such techniques can be applied effectively in the near future.

7 Conclusions

In this paper, we have presented a push-button formal verification technique to indirectly verify generators of multi-threaded code, that can be automatically performed each time the generator is applied. Besides its push-button nature, its main strength is its modularity, which is enabled by our earlier results [10, 47] on verifying generic constructs that are used to implement the communication between threads. This allows us to focus the technique proposed in the current paper to focus on individual threads in isolation. Due to this, it can verify the generation of thousands of lines of code in a few minutes. Furthermore, it can be easily adapted to changes, and made applicable to other DSLs and programming languages, as long as mappings to CFGs via our BIR language and the generation of separation logic specifications are constructed and updated accordingly.

Concerning future work, we have so far proven manually that transformations, such as the ones illustrated in Fig. 8, are correct. We will work on formally proving this using a theorem prover. Furthermore, as the current performance bottleneck is transition methods verification, we will work on the detection of functionally equivalent transitions, to reduce the amount of verification work. Furthermore, we will extend our method with support for the complete Slco language, and further extensions including timed behaviour [42, 43, 45]. Finally, we will also apply the same approach to other DSLs and programming languages, in particular for the generation of software for graphics processors.

References

1. Ab Rahim, L., Whittle, J.: Verifying semantic conformance of state machine-to-Java code generators. In: MODELS, Part I. LNCS, vol. 6394, pp. 166–180. Springer (2010)
2. Ab Rahim, L., Whittle, J.: A Survey of Approaches for Verifying Model Transformations. Software and Systems Modeling (2013), available online
3. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)

4. Amighi, A., Haack, C., Huisman, M., Hurlin, C.: Permission-based separation logic for multithreaded Java programs. Logical Methods in Computer Science **11**(1-2), 1–66 (2015)
5. Baldan, P., Corradini, A., Ehrig, H., Heckel, R., König, B.: Bisimilarity and Behaviour-preserving Reconfigurations of Open Petri Nets. In: CALCO. LNCS, vol. 4624, pp. 126–142. Springer (2007)
6. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development, Coq' Art: The Calculus of Inductive Constructions. Springer (2004)
7. Blech, J., Glesner, S., Leitner, J.: Formal Verification of Java Code Generation from UML Models. In: Fujaba Days 2005. pp. 49–56 (2005)
8. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: Verification of parallel and concurrent software. In: iFM. LNCS, vol. 10510, pp. 102–110. Springer (2017)
9. Bourke, T., Brun, L., Dagand, P.E., Leroy, X., Pouzet, M., Rieg, L.: A formally verified compiler for Lustre. In: PLDI. ACM SIGPLAN Notices, vol. 52, pp. 586–601. ACM (2017)
10. Bošnački, D., van den Brand, M., Gabriels, J., Jacobs, B., Kuiper, R., Roede, S., Wijs, A., Zhang, D.: Towards modular verification of threaded concurrent executable code generated from DSL models. In: FACS. LNCS, vol. 9539, pp. 141–160. Springer (2015)
11. Bunte, O., Groote, J., Keiren, J., Laveaux, M., Neele, T., de Vink, E., Wesselink, W., Wijs, A., Willemse, T.: The mCRL2 toolset for analysing concurrent systems: Improvements in expressivity and usability. In: TACAS, Part II. LNCS, vol. 11428, pp. 21–39. Springer (2019)
12. Chaki, S., Clarke, E., Groce, A., Jha, S., Veith, H.: Modular Verification of Software Components in C. In: ICSE. pp. 385–395. IEEE (2003)
13. Dalvandi, M., Butler, M., Rezazadeh, A.: From Event-B models to Dafny code contracts. In: FSEN. LNCS, vol. 9392, pp. 308–315. Springer (2015)
14. Dejanović, I., Vaderna, R., Milosavljević, G., Vuković, Ž.: TextX: A Python tool for Domain-Specific Languages implementation. Knowledge-Based Systems **115**, 1–4 (2017). https://doi.org/10.1016/j.knosys.2016.10.023
15. Ernst, D., Pfähler, J., Schellhorn, G., Haneberg, D., Reif, W.: KIV: Overview and VerifyThis competition. International Journal on Software Tools for Technology Transfer **17**(6), 677–694 (2015)
16. Groote, J., Jansen, D., Keiren, J., Wijs, A.: An O(m log n) algorithm for computing stuttering equivalence and branching bisimulation. ACM Transactions on Computational Logic **18**(2), 13:1–13:34 (2017)
17. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language LUSTRE. Proceedings of the IEEE **79**(9), 1305–1320 (1991)
18. Havender, J.: Avoiding deadlock in multitasking systems. IBM Systems Journal **7**(2), 74–84 (1968)
19. Holzmann, G.: The SPIN model checker: Primer and reference manual. Addison-Wesley Professional (2003)
20. Hülsbusch, M., König, B., Rensink, A., Semenyak, M., Soltenborn, C., Wehrheim, H.: Showing Full Semantics Preservation in Model Transformation - A Comparison of Techniques. In: IFM. LNCS, vol. 6396, pp. 183–198. Springer (2010)
21. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In: NFM, LNCS, vol. 6617, pp. 41–55. Springer (2011)
22. Jhala, R., Majumdar, R.: Software Model Checking. ACM Computing Surveys **41**(4), 21:1–21:54 (2009)

23. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Professional (2005)
24. Kumar, R., Myreen, M., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: POPL. ACM SIGPLAN Notices, vol. 49, pp. 179–191. ACM (2014)
25. Leino, K.: Dafny: a language and program verifier for functional correctness. In: LPAR. LNCS, vol. 6355, pp. 348–370. Springer (2010)
26. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7), 107–115 (2009)
27. Leroy, X.: Formal proofs of code generation and verification tools. In: SEFM. LNCS, vol. 8702, pp. 1–4. Springer (2014)
28. Müller, P., Schwerhoff, M., Summers, A.: Viper: A verification infrastructure for permission-based reasoning. In: VMCAI. LNCS, vol. 9583, pp. 41–62. Springer (2016)
29. Nicola, R.D., Vaandrager, F.: Action versus State Based Logics for Transition Systems. In: Semantics of Systems of Concurrent Processes, LITP Spring School on Theoretical Computer Science. LNCS, vol. 469, pp. 407–419. Springer (1990)
30. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer (2002)
31. Paige, R., Tarjan, R.: Three partition refinement algorithms. SIAM Journal on Computing **16**(6), 973–989 (1987)
32. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: SPIN'07. LNCS, vol. 4595, pp. 263–267. Springer (2007)
33. Pnueli, A., Shtrichman, O., Siegel, M.: The code validation tool CVT: Automatic verification of a compilation process. International Journal on Software Tools for Technology Transfer **2**(2), 192–201 (1998)
34. de Putter, S., Wijs, A.: A formal verification technique for behavioural model-to-model transformations. Formal Aspects of Computing **30**(1), 3–43 (2018)
35. de Putter, S., Wijs, A., Zhang, D.: The SLCO framework for verified, model-driven construction of component software. In: FACS. LNCS, vol. 11222, pp. 288–296. Springer (2018)
36. Reniers, M., Schoren, R., Willemse, T.: Results on embeddings between state-based and event-based systems. The Computer Journal **57**(1), 73–92 (2014)
37. Slind, K., Norrish, M.: A brief overview of HOL4. In: TPHOLs. LNCS, vol. 5170, pp. 28–32. Springer (2008)
38. Staats, M., Heimdahl, M.: Partial translation verification for untrusted code-generators. In: ICFEM. LNCS, vol. 5256, pp. 226–237. Springer (2008)
39. Stenzel, K., Moebius, M., Reif, W.: Formal Verification of QVT Transformations for Code Generation. In: MODELS. LNCS, vol. 6981, pp. 533–547. Springer (2011)
40. Sulzmann, M., Zechner, A.: Model checking DSL-generated C source code. In: SPIN. LNCS, vol. 7385, pp. 241–247. Springer (2012)
41. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model Checking Programs. Automated Software Engineering **10**(2), 203–232 (2003)
42. Wijs, A.: Achieving discrete relative timing with untimed process algebra. In: ICECCS. pp. 35–46. IEEE (2007)
43. Wijs, A.: What to do Next?: Analysing and Optimising System Behaviour in Time. Ph.D. thesis, VU University Amsterdam (2007)
44. Wijs, A., Engelen, L.: Efficient Property Preservation Checking of Model Refinements. In: TACAS'13. Lecture Notes in Computer Science, vol. 7795, pp. 565–579. Springer (2013)

45. Wijs, A., Fokkink, W.: From χ_t to μCRL: Combining performance and functional analysis. In: ICECCS. pp. 184–193. IEEE (2005)
46. Wijs, A., Neele, T., Bošnački, D.: GPUexplore 2.0: Unleashing GPU Explicit-state Model Checking. In: FM. LNCS, vol. 9995, pp. 694–701. Springer (2016)
47. Zhang, D., Bošnački, D., van den Brand, M., Huizing, C., B.Jacobs, Kuiper, R., Wijs, A.: Verifying atomicity preservation and deadlock freedom of a generic shared variable mechanism used in model-to-code transformations. In: MODELSWARD, Revised Selected Papers. CCIS, vol. 692, pp. 249–273. Springer (2016)