Embedding Formal Verification in Model-Driven Software Engineering with SLCO: An Overview

Anton Wijs (✉)

Eindhoven University of Technology, Eindhoven, The Netherlands
`a.j.wijs@tue.nl`

Abstract. In 2009, the Simple Language of Communicating Objects (SLCO) Domain-Specific Language was designed. Since then, a range of tools have been developed around this language to conduct research on a wide range of topics, all related to the construction of complex, component-based software, with formal verification being applied in every development step. In this paper, we present this range, and draw connections between the various, at first glance disparate, research results. We discuss the current status of the SLCO framework, i.e., the language in combination with the tools, and plans for future work.

Keywords: Domain-Specific Language, Model-Driven Software Engineering, formal verification, parallel software, component-based software

1 Introduction

The development of complex software, such as component-based software, is time-consuming and error-prone. One methodology aimed at making software development more transparent and efficient is *Model-Driven Software Engineering* (MDSE) [38]. In a typical MDSE workflow, software is (mostly automatically) constructed by first creating a high-level description of the system under development, by means of a *model*. Such a model is often expressed in a *Domain-Specific Language* (DSL). This initial model is subsequently gradually refined via *model transformations*, to add information to the model in a structured way, and finally, once the model is detailed enough, derive source code that implements the low-level description of the final model (see Figure 1). Such a workflow is also used in some *low-code application development* platforms [22].

Model transformations can be viewed as artefacts that accept a model as input, and either produce a new model (model-to-model) or code (model-to-code) as output.[1] Once defined, they can be applied automatically on models. Ideally, once the initial model has been created, and the necessary model transformations identified or designed, the MDSE procedure is fully automatic, resulting in source code that exactly implements the intended functionality, or at least requires only minor manual alteration.

[1] Model-to-code transformations are also known as code generators.

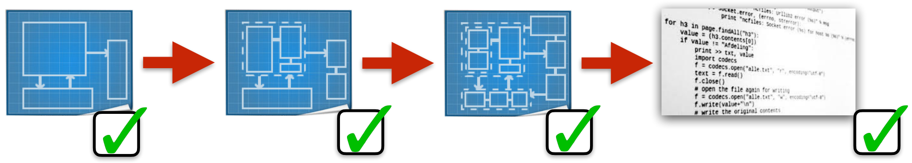


Fig. 1: Verification in a Model-Driven Software Engineering workflow.

Automatically developing software via MDSE goes a long way in reducing the introduction of errors and making software development more efficient. However, functional correctness of the developed software is not guaranteed. About 15 years ago, researchers in the Software Engineering & Technology group at the Eindhoven University of Technology, started to investigate in which ways software verification techniques could be embedded in MDSE in a seamless way [7, 25]. After a collaboration with industry, it soon became clear that in order to structurally perform this research, the starting point needed to be a DSL that is relatively simple, yet expressive enough to model the basic functionality of software consisting of multiple interacting components. This lead to the creation of the *Simple Language of Communicating Objects* (SLCO) [5].

The development of SLCO and its model transformations was originally motivated by research questions addressing the internal and external quality of model transformations. The internal quality refers to the definition of a model transformation, while the external quality considers the process of applying a transformation on a model [4]. By analysing the impact of a model transformation on a given SLCO model, or the potential impact of a transformation on an arbitrary SLCO model, the external quality is assessed. Soon, however, SLCO was used for research on embedding formal verification techniques throughout the entire MDSE workflow, so that not only the initial model, but all produced artefacts can be formally verified (see the green ticks in Figure 1).

In this paper, we present an overview of the research conducted in the last 15 years with SLCO on formal verification techniques to verify the various MDSE artefacts. While the individual results have already been published, such an overview allows viewing the bigger picture, and the directions in which the research as a whole is going in the future.

2 A DSL for Component-Based Software

In 2009, SLCO version 1.0 was developed to address a particular case study, namely the generation of Not Quite C (NQC) code for a controller of a conveyor belt built in Lego Mindstorms [6, 7]. The key part of this platform is a programmable controller called RCX. It has an infrared port for communication and is connected by wires to sensors and motors for environment interaction. This imposes particular restrictions, such as the fact that communication is asynchronous and unreliable, i.e., messages may get lost. These restrictions in-

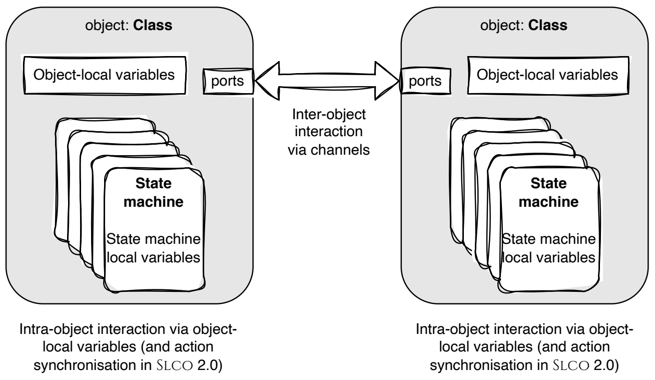

object: **Class**
Object-local variables
ports
Inter-object interaction via channels
ports
object: **Class**
Object-local variables
State machine
State machine local variables
State machine
State machine local variables
Intra-object interaction via object-local variables (and action synchronisation in Slco 2.0)
Intra-object interaction via object-local variables (and action synchronisation in Slco 2.0)

Fig. 2: The general structure of an Slco model.

fluenced the design of Slco 1.0. For instance, besides reliable synchronous and asynchronous channels, Slco also has an unreliable asynchronous channel as a building block. In addition, as NQC does not support arrays, these were also not added to Slco 1.0. Finally, as NQC allows working with timers, Slco was equipped with a delay statement, to allow expressing that a component should wait a specified number of milliseconds.

The general structure of an Slco model is presented in Figure 2. In a model, one or more classes are defined, which can be instantiated as objects. In a class, local variables are defined, which can be of one of the primitive types: Boolean, Integer or String. In addition, a class has a finite number of state machines. Each state machine contains a finite number of states, one of which being the *initial* state, and transitions, and optionally local variables of the primitive types. Finally, a class possibly has one or more *ports*, to which channels can be connected.

Within an object, i.e., a class instance, the state machines can interact via the variables defined at the object level. State machines in different objects can communicate via channels. In this way, Slco can be used both for the specification of shared-memory, parallel systems and distributed systems.

Channels are connected between the ports of their respective objects. Messages sent over these channels contain a *signal*, i.e., a header, and a fixed number of values, each of a primitive type. When a channel instance is created, the type of the messages for this channel is defined, and it is specified whether the channel is synchronous or asynchronous, and in the latter case, what the size of its FIFO buffer is, and whether it is lossless or lossy.

State machines can exhibit behaviour. At any time, one of the states of a state machine is its *current* state, which initially is the initial state. If an outgoing transition of this current state is *enabled*, the state machine can fire the transition and move to the target state of that transition. With each transition, a list of zero or more *statements* are associated. If a transition has no statements, it is always enabled. If it has at least one statement, it is enabled iff its first statement is enabled. Firing a transition means considering each associated statement, in the order defined by the list, for execution. In SLCO 1.0, the following statement types are available:

– *Assignment* (`x := E`): assign to variable `x` the value defined by the expression `E`. Variable `x` can be any variable in the scope of the state machine, i.e., it can be either state machine local or local to the object containing the state machine. In expression `E`, references to variables in the scope of the state machine and constants can be combined with the usual operators. The expression must evaluate to a value of the same type as `x`. Assignments are always enabled.
– *(Boolean) Expression*: If an expression `E` evaluates to a Boolean value, it can be used as a stand-alone statement, and act as a guard. Such a statement is enabled iff it evaluates to **true**.
– *Send* (`send <message> to <port>`): send the given message via the channel connected to `<port>`. In case the channel is synchronous, this statement is enabled iff at least one other state machine can receive the message via the other port of the channel. If the channel is asynchronous, the statement is enabled iff the buffer of the channel is not yet full. When fired, the message is either sent to a receiving state machine (synchronous) or added to the FIFO buffer of the associated channel (asynchronous).
– *Receive* (`receive <message>|<guard> from <port>`): if there is a message available to be received, its signal matches the one specified, and the (Boolean expression) `guard`, which may refer to the message to be received via the variable(s) in which the message value(s) is/are to be stored, evaluates to **true**, then the receive statement is enabled, and when executed, results in the message of the sender being received (synchronous) or the first message in the FIFO buffer of the channel being received and removed from the buffer (asynchronous).
– *Delay* (`after <time> ms`): wait for `time` milliseconds. This statement is always enabled.

Each statement is *atomic*, i.e., its execution cannot be interrupted. Regarding concurrency, SLCO has an interleaving semantics.

The fact that a transition can have a list with more than one statement may lead to situations in which a transition is fired, but its execution cannot terminate, due to the execution reaching a statement that is not enabled. For instance, the sequence `x := 0; y := 1; x = y` cannot terminate, unless another state machine interferes with `x` and `y` to make the expression `x = y` evaluate to **true**. If the execution of a transition cannot terminate, the state machine is stuck in an intermediate state, in-between the source and target states of the fired transition. To make it simpler to reason about this, a fragment of SLCO is referred to as 'simple SLCO', which only differs from SLCO in the fact that at most one statement can be associated with each transition. A model-to-model transformation has been defined, that can transform SLCO models to semantically equivalent simple SLCO models, by introducing additional states and transitions where needed.

```
 1 model ex_chan {
 2   classes
 3     P {
 4       variables
 5         Integer x
 6       ports Out
 7       state machines
 8         SM1 {
 9           initial R0 states R1
10           transitions
11             R0 -> R1 { x = 0; x := 1 }
12             R1 -> R0 { send M(x) to Out }
13         }
14     }
15 
16     Q {
17       variables
18         Integer result
19       ports In
20       state machines
21         SM2 {
22           initial S0
23           transitions
24             S0 -> S0 { receive M(result | result % 2 = 1) from In }
25         }
26     }
27   objects p: P(), q: Q()
28   channels c (Integer) sync between p.Out and q.In
29 }
```

Fig. 3: An Slco model of a distributed system.

Finally, before we discuss the research conducted with Slco, we present an example Slco model in Figure 3. The name of the model is defined at line 1. Furthermore, classes `P` and `Q` are defined at lines 3–26, and instantiated to objects `p` and `q` at line 27. In each class, variables and ports are defined (lines 4–5 and 17–18, and lines 6 and 19, respectively). Each class contains one state machine. The states of these state machines are defined at lines 9 and 22, and their transitions are listed at lines 10–12 and 23–24. Finally, a synchronous channel between the ports of objects `p` and `q` is defined at line 28. Note that `q` can successfully receive one message from `p`, as the sent message has a matching signal `M` and contains the value `1`, which meets the requirement of `q` that the value must be odd. Also, only a single message can be sent, since after sending, state machine `SM1` returns to state `R0`, at which point execution is permanently blocked, since the expression `x = 0` evaluates to **false**, and once set to `1`, `x` is never set to `0` again.

3 Verifying Model-to-Model Transformations

3.1 Reverification of Models

To investigate the ability to reason about the external quality of model-to-model transformations, a number of model-to-model transformations were developed for Slco, using the Xtend and Atl model transformation languages and the

Xpand tool [10, 35], to be applied in the Lego Mindstorms conveyor belt case study. Some of these addressed refactoring aspects, such as the automatic removal of unused variables, channels and classes, changing object-local variables to state machine local ones in case they are only accessed by a single state machine, and merging objects together into a single object. The latter also affects interaction between state machines: as state machines from different objects are moved to the same object, any interaction via channels between them is transformed to interaction via shared variables. For Lego Mindstorms, this model transformation allowed to meet the criterion that the number of objects in the final model has to match the number of RCXs in the system setup.

In addition, model-to-model transformations affecting the semantics were defined, such as a transformation that introduces delays in a given set of transitions, a transformation that achieves synchronous communication with asynchronous channels, a transformation that achieves broadcasting messages between more than two state machines via a number of channels, a transformation that introduces the Alternating Bit Protocol (ABP) [26] to deal with lossy channels, and a transformation that makes the sender of a message explicit in each message.

Finally, a number of model transformations were defined to transform Slco models to artefacts written in other languages, such as a transformation to Promela to allow the model checking of Slco models with the Spin model checker [32], a transformation to `dot` to allow the visualisation of state machines, and, of course, a code generator for NQC to produce source code.

The model-to-model transformation from Slco to Promela was used in a first attempt to verify model-to-model transformations [6]. Since verifying the model-to-model transformations themselves would require new verification techniques, the approach was to verify a given Slco model, and reverify it each time a model-to-model transformation that produced a refined Slco model had been applied to it. This approach does not verify that a given model-to-model transformation is guaranteed to produce correct models in general, but at least it allows to verify that it works correctly on a case-by-case basis. After every transformation application, the resulting Slco model was transformed to a Promela model, to be verified with Spin.

However, a major drawback of this method is its limited scalability. Table 1 shows the impact of model-to-model transformations on the size of the state space of a simple Slco model with a producer and a consumer object, i.e., a model very similar to the one of Figure 3, in which one state machine sends messages and another one receives those messages [6, 25]. Changing the initially synchronous channels to asynchronous ones doubles the size of the state space, but making this channel lossy, and introducing the ABP protocol has a significant impact on the state space size. Finally, adding delays to the transitions further increases the state space by a factor 10. Considering that this is only a model with a single channel, one can imagine the impact on models with many more channels. Because of this state space explosion, and the fact that model-to-model transformations by themselves are actually relatively small and typically only impact a particular part of a model, the ambition was soon formulated to conduct

Table 1: State space sizes of models specifying a producer and a consumer.

Model	**# States**	**# Transitions**
Synchronous	4	6
Asynchronous	8	11
Lossy + ABP	114,388	596,367
Delays	1,009,856	5,902,673

research on verifying model-to-model transformation definitions themselves, and reason about their impact on models in general.

3.2 Direct Verification of Model-to-Model Transformations

In 2011, research was started on directly verifying the impact of model-to-model transformations on models in general [60,61,64]. Contrary to the majority of the work on model transformation verification at that moment [3,55], which focussed on wellformedness of transformations, i.e., that model transformations produce syntactically correct output, and questions such as whether a model transformation is terminating and/or confluent, we decided to focus on the semantical guarantees that model-to-model transformations can provide.[2] In particular, since in model checking, models are checked w.r.t. given functional properties formalised in temporal logic, we were interested in verifying whether model-to-model transformations preserve those properties. Being able to conclude this would mean that reverification of models would no longer be needed.

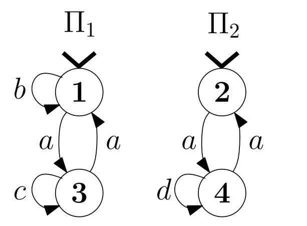
Inspired by action-based model checking, we decided to reason about the semantics of SLCO models by means of *Labelled Transition Systems* (LTSs), as defined in Definition 1.

Definition 1 (Labelled Transition System). *A* Labelled Transition System *L is a tuple $\langle S, A, T, \hat{s} \rangle$, with*

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

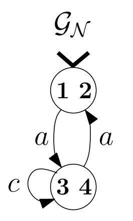
SLCO has a formal semantics, and it was straightforward to map that semantics to LTSs. Actually, since model-to-model transformations for component-based systems tend to transform individual components, we reasoned about the semantics of component-based systems by means of *LTS networks* [41]. In such a network, the potential behaviour of each individual component is represented by an LTS, and the potential interaction between these components is defined by a set of *synchronisation laws* $\mathcal{V}$ that expresses which actions of the individual LTSs need to synchronise with each other, and which do not. An LTS network semantically corresponds with an individual *system LTS*, in which the potential behaviour of the components is interleaved, and synchronisations are applied

[2] Other works addressing the semantical impact of transformations include [28,33,46].



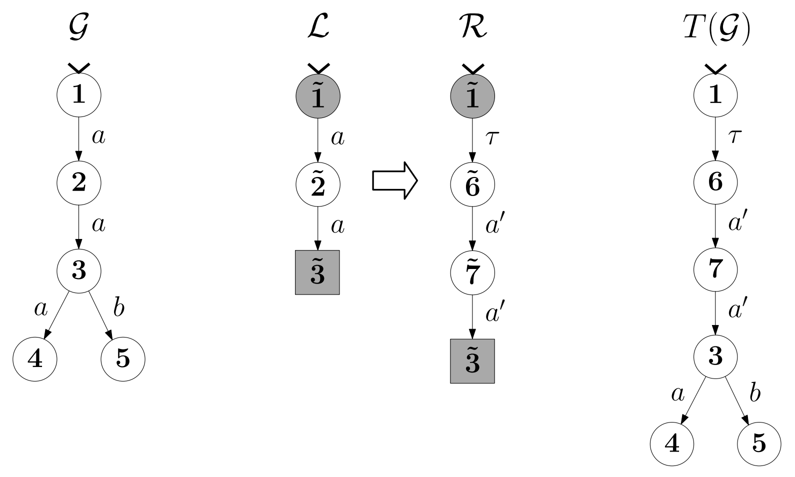
$$\mathcal{V} = \left\{ \begin{array}{l} (\langle a, a\rangle, a) \\ (\langle b, d\rangle, e) \\ (\langle c, \bullet\rangle, c), \end{array} \right\}$$

(a) An LTS network $\mathcal{N}$.



(b) The system LTS of $\mathcal{N}$.

Fig. 4: An example LTS network and its corresponding system LTS.



(a) Input LTS. (b) Transformation rule. (c) Output LTS.

Fig. 5: An example of applying an LTS transformation rule on an LTS. Round grey state: glue in-state, square grey state: glue exit-state.

where needed. Figure 4 presents an example LTS network (Figure 4a) and its corresponding system LTS (Figure 4b): How LTSs Π_1 and Π_2 should (not) synchronise is given by the three laws in $\mathcal{V}$: action a needs to be performed by both LTSs together, leading to an a-transition in the system LTS, b and d of Π_1 and Π_2, respectively, need to synchronise to an e-transition in the system LTS, and c can be performed by Π_1 independently.

In this setting, model-to-model transformations can be formalised by means of *LTS transformation rule systems*, inspired by double pushout graph rewriting [24]. Here, we explain the basics by means of an example. A formal treatment can be found in [51, 60]. Figure 5 shows an example LTS $\mathcal{G}$ on the left (Figure 5a), with state 1 the initial state. A transformation rule is a pair of LTSs $\langle \mathcal{L}, \mathcal{R} \rangle$, with $\mathcal{L}$ and $\mathcal{R}$ having some states in common, called the *glue in-states* and *glue exit-states*. In the example, the round grey states are glue in-states and the square grey states are glue exit-states. The example rule (Figure 5b) expresses the following: a sequence of two a-transitions should be replaced by a τ-transition, followed in sequence by two a'-transitions. Moreover, the rule can only be matched on a sequence of two a-transitions in $\mathcal{G}$ if that sequence satisfies the following criteria:

1. The first state does not have any additional outgoing transitions (expressed by the glue in-state of $\mathcal{L}$);
2. The last state does not have any additional incoming transitions (expressed by the glue exit-state of $\mathcal{L}$);
3. The intermediate state has no additional in- or outgoing transitions (expressed by the fact that state $\tilde{2}$ of $\mathcal{L}$ is not present in $\mathcal{R}$, meaning that a state matched on $\tilde{2}$ is supposed to be removed and replaced by two new states $\tilde{6}$ and $\tilde{7}$, plus the fact that the removal of states in an LTS may not affect its transitions in ways not addressed by the transformation rule).

The rule $\langle \mathcal{L}, \mathcal{R} \rangle$ is applicable on $\mathcal{G}$, and applying it results in the LTS $T(\mathcal{G})$ given in Figure 5c.

Model-to-model transformations formalised by means of LTS transformation rule systems can be reasoned about without considering a particular LTS on which they are applied. We only assume that such an LTS satisfies particular functional properties of interest. For instance, if an LTS satisfies the property "Always eventually c occurs", expressed in an action-based temporal logic, then it is clear that the transformation rule given in Figure 5 preserves this property when applied on that LTS, regardless of the latter's structure. The property is preserved as the rule does not transform c-transitions, nor does it affect the reachability of states in the LTS.

The model-to-model transformation verification technique developed in [50, 51, 62] considers functional properties expressed in the modal μ-calculus [39]. Given a μ-calculus formula φ, actions in $\mathcal{L}$ and $\mathcal{R}$ of a transformation rule are automatically abstracted away, i.e., replaced by the silent action τ, if they are considered irrelevant for φ [45]. After this, if $\mathcal{L}$ and $\mathcal{R}$, extended to make explicit that they represent embeddings in a larger LTS on which they are applied, are *divergent-preserving branching bisimilar* [29], which is an equivalence for LTSs sensitive to τ-transitions while still considering the branching structure and τ-loops of the LTSs, then it can be concluded that φ will be satisfied by the LTS produced by the transformation. Although highly non-trivial, this can be extended to LTS transformation rule systems that transform the synchronising behaviour between LTSs. In [51], a formalisation of this technique is presented in detail that has been proven correct with the Coq Proof Assistant [9].

Performance-wise, model-to-model transformation verification is a great improvement over reverifying models. All the considered example transformations could be verified w.r.t. property-preservation practically instantly [51]. The approach has one main drawback, though: sometimes, functional properties can only be expressed, and only become relevant, once the model has obtained a certain amount of detail. If a property must be expressed about the behaviour introduced by one or more model-to-model transformations, property-preservation is not relevant, and verification of the current model seems inevitable.

To possibly even avoid reverification in such cases, more recently, we conducted research on reasoning about the *effect* of an individual LTS transformation rule r on a system property ψ [21], expressed in Action-based Linear Temporal Logic (ALTL) [27, 49], when applied on a component satisfying that property. First, a representative LTS $\mathcal{L}_\psi$ for components satisfying an ALTL formula ψ is constructed, by creating the cross-product of a representative LTS $\mathcal{L}$ for all LTSs on which r is applicable with an action-based Büchi automaton $\mathcal{B}_\psi$ encoding ψ [18]. On $\mathcal{L}_\psi$, r is applied, resulting in action-based Büchi automaton $T(\mathcal{L}_\psi)$. After detecting and removing non-accepting cycles of $T(\mathcal{L}_\psi)$, and minimising the resulting Büchi automaton using standard minimisation techniques [23], a characteristic formula for the action-based Büchi automaton is created in the form of a system of μ-calculus equations. Similar to property-preservation checking, this approach works practically instantly. To generalise it to the setting of [51], in future work, rule systems consisting of multiple LTS transformation rules should be considered, and functional properties written in the modal μ-calculus, more expressive than ALTL, should be supported. This approach is promising, but still restricted to updating temporal logic formulae expressed originally for the initial model. If entirely new properties become relevant for a model at a later stage in the MDSE workflow, verification of that model is unavoidable.

Challenges and directions for future work The developed technique to formally verify model-to-model transformations reasons about the semantics of component-based systems by focussing on LTSs and their transformation, but a suitable formalism to express SLCO-to-SLCO transformations in a way compatible with this is yet to be identified. One direction for future work is to find such a model transformation language. Existing general-purpose transformation languages, such as ATL and XTEND, may be suitable, or a Domain-Specific Transformation Language could be developed that directly involves the SLCO constructs. Extending the technique to reason about the effect of a transformation on a system property will be further investigated as addressed above.

4 Verifying Code Generators

When applied on an SLCO model, a code generator should produce code that is (as much as possible) semantically equivalent to the model. In 2014, a new project started, focussed on the model-driven development of multi-threaded

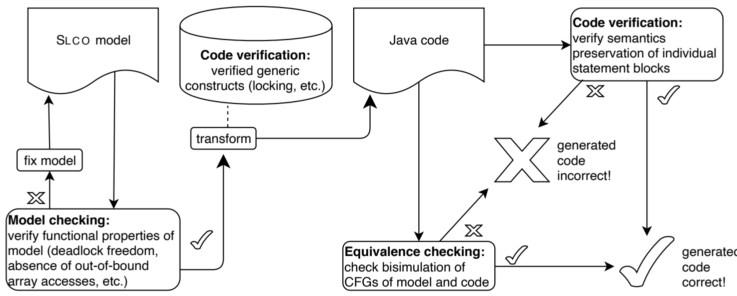

SLCO model
Code verification: verified generic constructs (locking, etc.)
Java code
Code verification: verify semantics preservation of individual statement blocks
transform
fix model
generated code incorrect!
Model checking: verify functional properties of model (deadlock freedom, absence of out-of-bound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

Fig. 6: Verified generation of multi-threaded JAVA code.

software. It was decided to develop a code generator for multi-threaded JAVA code. JAVA being more elaborate than NQC, it was soon clear that SLCO needed to be extended. Version 2.0 of the language [53] introduced the following features:

– Support for the Byte primitive type, and arrays of the primitive types;
– Transition priorities. These allow expressing that the outgoing transitions of a state must be considered for firing in a fixed order;
– An *Action* statement (`do <action>`): this allows assigning arbitrary action labels to transitions, which can represent particular events, such as calling external functions. Action statements are always enabled.
– A *Composite* statement (`[<expr>;<assgn_1>;...;<assgn_n>]`): this allows combining certain statements into a single, atomically executed, statement. It starts with an expression, which may be **true**, and a sequence of one or more assignments. The statement is enabled iff the expression is enabled.

The general workflow of producing multi-threaded JAVA code in a verified way is presented in Figure 6, while an example SLCO model and part of its JAVA implementation is given in Figure 7. The SLCO-to-JAVA code generator creates one thread for each state machine in the objects of the given SLCO model. In Figure 7, some of the code for a thread executing the transitions of state machine `SM1` of object `p` is given. Each thread has access to a *lock keeper* (lines 6–7 of the JAVA code), which manages the locks used to avoid data races when accessing object-local variables. This lock keeper uses an ordered locking scheme that prevents circular lock dependencies between threads.

A thread executes according to the associated state machine as follows: the initial state is defined by the constructor method at line 10. In the `exec` method (line 26), the current state of the state machine is repeatedly checked (lines 27–28), and depending on its value, one or more functions are executed that correspond one-to-one with a transition in the SLCO model. At lines 16–22, the function `execute_R0_0` is given, which corresponds with the transition at line 11 of the SLCO model. First, it is attempted to acquire a lock for variable `x`. Once this is achieved, it is checked whether the expression of the composite statement evaluates to **true** (line 18). If it does not, the lock is released and **false** is returned. If it does, `x` is updated, the lock released, and **true** is returned. If

```
model ex_shr {
  actions a
  classes
    P {
      variables
        Integer x
      state machines
        SM1 {
          initial R0 states R1
          transitions
            R0 -> R1 { [x = 0; x := 1] }
            R1 -> R0 { do a }
        }
        SM2 {
          initial S0
          transitions
            S0 -> S0 { x % 2 = 1 }
        }
    }
  objects p: P()
}
```

```
...
class java_SM1Thread extends Thread {
  private Thread java_t;
  // Current state
  private ex_shr.java_State java_cState;
  // Keeper of global variables
  private ex_shr.java_Keeper java_kp;

  // Constructor
  java_SM1Thread (ex_shr.java_Keeper java_k) {
    java_cState = ex_shr.java_State.R0;
    java_kp = java_k;
  }

  // Transition functions
  boolean execute_R0_0() {
    // [ x = 0; x := 1 ] ... Acquire locks ...
    if (!(x == 0)) { java_kp.unlock(1); return false; }
      x = 1;
      java_kp.unlock(1);
      return true;
  }
  boolean execute_R1_0() { a(); return true; }

  // Execute method
  public void exec() {
    while(true) {
      switch(java_cState) {
        case ex_shr.java_State.R0:
          if (execute_R0_0()) { java_cState = ex_shr.java_State.R1; }
          break;
        case ex_shr.java_State.R1:
          if (execute_R1_0()) { java_cState = ex_shr.java_State.R0; }
          break;
        default: return;
  }}}
...
```

Fig. 7: An SLCO shared memory system (top) and derived JAVA code (bottom).

a transition function returns **true**, the thread updates its state and continues checking the current state. Note at line 23 that the action `a` is mapped to some external function with the same name.

Complete formal verification of a code generator is very challenging [72]. First, we focussed on proving correctness of the *model-independent* parts of the code: we proved that the lock keeper does not introduce deadlocks due to threads waiting for each other [73], that the JAVA channels work as specified by the SLCO channels [16], and that a safety construct called *Failbox* works as intended [15]. For this, the VERIFAST code verifier was used, which allows verifying that JAVA code adheres to pre- and post-conditions specified in separation logic [34]. These verified constructs can be safely used in generated code (see Figure 6).

The next step was to verify *model-specific* code. Verifying that a code generator always produces correct model-specific code would require reasoning about all possible inputs, i.e., SLCO models. As this is very complex, we focussed on trying to verify automatically that for a given SLCO model, the produced JAVA code correctly implements it, i.e., adheres to the semantics of the model. We achieved this in a two-step approach [67]: first, the control flows of both a thread and its corresponding state machine are extracted. After some straightforward transformations that bring the two control flow graphs conceptually closer together, they are stored in a common graph structure. It is then checked whether those graphs are *bisimilar*. If they are, then we have established that the thread and the state machine perform their steps in equivalent ways. What remains is to establish that the individual steps of the thread indeed correspond with the individual steps of the state machine. For this, code verification is used again. The individual JAVA transition functions are automatically annotated with pre- and post-conditions in separation logic, expressing the semantics of the corresponding SLCO statements. This time, we used the VERCORS verifier to perform the verification [12]. As the pre- and post-conditions are generated automatically, performing the verification only requires pushing a button.

Finally, we investigated techniques to check whether an implementation would still adhere to SLCO's semantics if a platform with a *weak memory model* was targeted [52]. Such a model allows out-of-order execution of instructions, which may violate the intended functionality. In related work, this problem has been addressed in two different ways: in one, a dependency graph is constructed by statically analysing the code [2, 57]. This graph encodes which instructions depend on each other due to them accessing the same variables. Next, cycles in this graph that meet certain criteria, depending on the targeted memory model, represent violations of that model. The other way is to apply model checking, considering both the usual possible executions, with instructions occurring in the specified order, and executions in which the instructions have been reordered, insofar allowed by the memory model [1, 44]. The drawback of the first approach is its imprecision, while the drawback of the second approach is a state space explosion that is typically even much worse than in standard model checking.

Our contribution was to combine the two approaches: first, explore the state space of the SLCO model, but only considering the executions with instructions

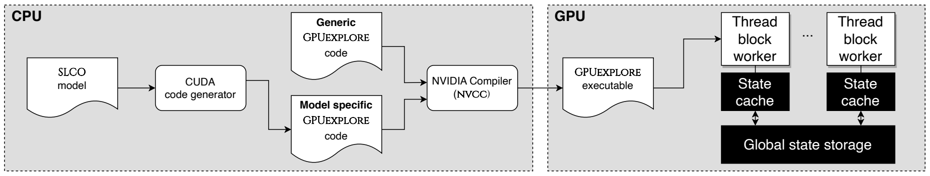


Fig. 8: The workflow from Slco model to GPUexplore model checking.

in the specified order, and derive from this a dependency graph. Second, apply cycle detection analysis on this dependency graph. For the state space exploration step, a model-to-model transformation from Slco to mCRL2 [19] was devised. As the produced graphs tend to be more precise than when using static analysis, the results in our experiments were of higher quality, and the overall runtime was often even faster. As the number of elementary cycles in a graph can grow exponentially, constructing a more accurate dependency graph can avoid introducing many cycles that an over-approximation of the potential behaviour would introduce. This reduced number of cycles greatly impacts the processing time, often compensating for the time it takes to explore the state space.

Challenges and directions for future work The main challenge in this research line is to achieve full verification of code generators. In related work on compiler and code generator verification, full verification has been achieved with theorem proving [11, 14, 40, 43, 58], but this is a labour-intensive approach that is not very flexible w.r.t. updates of the compiler or generator. We plan to work on techniques that allow flexible maintenance of correctness proofs.

Another direction currently investigated involves the generation of code for *graphics processing units* (GPUs) [30]. For many-core programs, however, Slco is not directly suitable. *Array languages*, on the other hand, have been designed with parallel array processing in mind, which aligns very well with typical GPU functions. We are currently investigating how to embed program verification into the Halide language, a language to express image and tensor computations [54]. This language separates *what* a program should do, i.e., its functionality, from *how* it should do it, i.e., the scheduling that involves performance optimisations. Besides making development more insightful, this separation also positively affects verifiability. Verification of the functional correctness of a program can be separated from verifying that optimisations applied to it preserve that correctness. In other work, we focus on updating pre- and post-conditions when code is automatically optimised, to allow for push-button reverification of the code [56].

5 GPU-Accelerated Model Checking

We addressed the verification of model-to-model transformations and code generators, but proving their correctness ultimately depends on the input models being correct. Hence, verifying the correctness of Slco models cannot be avoided, and sometimes needs to be performed multiple times in an MDSE workflow, as discussed in Section 3.2. Initially, we developed an Slco-to-mCRL2 transformation for this, to apply the mCRL2 toolset [19] for the model checking of (untimed) Slco models. Recently, we integrated Slco in another line of research that started in 2013, focussed on accelerating model checking with GPUs [20,47,69–71].

The research on GPU-acceleration of model checking is motivated first of all by the fact that for a seamless integration of formal verification in MDSE, it is crucial that models can be verified efficiently. If verification takes a long time, this hinders development. Second of all, as hardware developments are increasingly focussed on dedicated devices such as GPUs and adding cores to processors, as opposed to making individual cores faster, computationally intensive computations, such as model checking, require massively parallel algorithms [42].

Figure 8 presents the workflow of formally verifying the correctness of Slco models with the model checker GPUexplore version 3.0 [65,66]. First, an Slco model is analysed by a CUDA code generator. CUDA, the Compute Unified Device Architecture, is a parallel computing platform and application programming interface developed by NVIDIA, that can be used to develop programs for their GPUs. The generator produces CUDA C++ code that implements an explicit-state model checker for that specific Slco model: it includes generated functions that allow the evaluation and firing of Slco transitions by directly executing instructions corresponding with the associated Slco statements.

The generated code can be compiled with NVIDIA's NVCC compiler. Note in Figure 8 that the compiler combines generic, model-independent code, with model-specific code, similar to the Slco-to-Java transformation (Section 4).

On the right of Figure 8, the main concepts of a GPUexplore program are mapped to a GPU architecture: A GPU consists of many streaming multiprocessors (SM) that each have a limited amount of fast, on-chip *shared* memory, and one shared pool of *global* memory. Typically, a GPU program consists of a program, executed by one or more CPU threads, in which GPU functions, called *kernels*, are launched. These kernels are typically executed by many thousands of threads simultaneously. Threads are grouped into *blocks*. A block is executed by an SM, and the threads in a block share a specified amount of shared memory. It is not possible for the threads in one block to access the shared memory of another block. Finally, all blocks share the global memory. In GPUexplore, this memory is used to maintain a large hash table, in which the Slco model states are stored as they are reached, starting with the initial state.

An Slco model state is a vector defining a state of the model, i.e., it defines for each state machine its current state, and for each variable its current value. Each block repeatedly fetches *unexplored* states, i.e., states for which the outgoing transitions of the corresponding current states of the state machines have not yet been considered for firing. Exploring these states leads to the creation of *successors*, i.e., states reachable by firing a transition. This is conducted in parallel by the threads in a block; GPUexplore runs blocks of 512 threads each. Successors are temporarily stored in shared memory, in which a block-local hash

Table 2: State space exploration speed of Spin, LTSmin and GPUexplore, in millions of states per second. -o.m.-: out of memory (32 GB).

Model	Nr. states	Spin 4-core	LTSmin 4-core	GPUexplore
`adding.50+`	529,767,730	-o.m.-	5.36	**148.28**
`anderson.6`	18,206,917	1.36	1.31	**31.57**
`at.6`	160,589,600	0.87	2.39	**40.56**
`frogs.5`	182,772,126	1.05	2.63	**10.31**
`lamport.8`	62,669,317	1.78	2.19	**34.92**
`peterson.6`	174,495,861	0.76	2.45	**33.58**
`szymanski.5`	79,518,740	1.57	1.82	**18.34**

table is maintained. This prevents blocks from frequently accessing slow global memory (which is typically a major performance bottleneck). Once a batch of new successors has been generated, their presence in the global memory hash table is checked. States not yet present are added, ready to be explored in the next round. This procedure is repeated until no more states are generated.

Currently, GPUexplore supports deadlock checking, with support for the verification of Linear Temporal Logic (LTL) [49] formulae being planned for the near future. Table 2 presents some results obtained when comparing the state space exploration speed of GPUexplore with Spin and the model checker LTSmin [36]. Both Spin and LTSmin support CPU multi-core explicit-state model checking. The models listed here are all Slco models obtained by translating the model in the Beem benchmark suite [48] of the same name from the DVE language to Slco, except for `adding.50+`, which was obtained by scaling up the `adding` models present in that benchmark suite. We used a machine with a four-core CPU i7-7700 (3.6 GHz), 32 GB RAM, and an NVIDIA Titan RTX GPU with 24 GB global memory, running Linux Mint 20 and CUDA 11.4.

As LTSmin achieves near-linear speedups as the number of used cores is increased [59], these numbers indicate how many cores would be needed to match the speed of GPUexplore. GPUexplore can reach impressive speeds up to 148 million states per second. However, what stands out is that the achieved speed differs greatly between models, more than with Spin and LTSmin. In the near future, we will inspect the models and their state spaces, to identify the cause for these differences, and improve the reliability of GPUexplore.

Challenges and directions for future work The first aspect to address is the verification of temporal logic formulae. First, we will focus on LTL. However, state-of-the-art sequential LTL verification algorithms rely on Depth-First Search (DFS) of the state spaces, as they involve cycle detection. Since DFS is not suitable for GPUs, GPUexplore applies a greedy, Breadth-First Search based exploration algorithm, in which cycle detection cannot be integrated as straightforwardly. In the past, we have investigated algorithms for this that are incomplete [68]. Designing an alternative that is complete remains a challenge.

Another line of research is to achieve GPU acceleration of *probabilistic* model checking [8]. In the past, this has been partially accelerated with GPUs [13, 17, 37, 63]: once the state space has been generated, verification of a probabilistic property, formalised in Probabilistic Computation Tree Logic [31], involves repeated matrix-vector multiplications, which GPUs can perform very rapidly. Also accelerating the state space generation will likely be a major step forward.

6 Conclusions

We presented an overview of the research conducted in the last decade on integrating formal verification into an MDSE workflow centered around the Slco DSL. For an effective integration, efficient verification of models, model-to-model transformations and code generators is crucial. In the three research lines focussing on each of these three types of MDSE artefacts, important steps have been made, and open challenges remain for the (near) future.

One particular challenge bridging the first two lines concerns identifying ways to combine model verification and model-to-model transformation verification, ideally to achieve an automatic verification technique that, depending on the property, the model, and the transformation to be applied, can derive how the transformed model relates to that property. We envision that in order to derive this, a number of verification results need to be established, of which some could possibly be determined via model verification, while for others, model-to-model transformation verification could be more efficient.

References

1. Abdulla, P.A., Atig, M.F., Ngo, T.P.: The best of both worlds: Trading efficiency and optimality in fence insertion for TSO. In: ESOP. LNCS, vol. 9032, pp. 308–332. Springer (2015)
2. Alglave, J., Kroening, D., Nimal, V., Poetzl, D.: Don't sit on the fence: a static analysis approach to automatic fence insertion. ACM Trans. on Progr. Lang. and Syst. **39**(2), 6 (2017)
3. Amrani, M., Combemale, B., Lúcio, L., Selim, G., Dingel, J., Le Traon, Y., Vangheluwe, H., Cordy, J.: Formal Verification Techniques for Model Transformations: A Tridimensional Classification. Journal of Object Technology **14**(3), 1–43 (2015). https://doi.org/10.5381/jot.2015.14.3.a1
4. van Amstel, M.: Assessing and Improving the Quality of Model Transformations. Ph.D. thesis, Eindhoven University of Technology (2011)
5. van Amstel, M., van den Brand, M., Engelen, L.: An Exercise in Iterative Domain-Specific Language Design. In: EVOL/IWPSE. pp. 48–57. ACM Press (2010)
6. van Amstel, M., van den Brand, M., Engelen, L.: Using a DSL and Fine-Grained Model Transformations to Explore the Boudaries of Model Verification. In: MVV. pp. 120–127. IEEE Computer Society Press (2011)
7. van Amstel, M., van den Brand, M., Protić, Z., Verhoeff, T.: Model-Driven Software Engineering. In: Automation in Warehouse Development, chap. 4, pp. 45–58. Springer (2011)
8. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
9. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development, Coq' Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science, Springer (2004)

10. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend - Second Edition. Packt Publishing (2016)
11. Blech, J., Glesner, S., Leitner, J.: Formal Verification of Java Code Generation from UML Models. In: Fujaba Days 2005. pp. 49–56 (2005)
12. 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)
13. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.: Parallel Probabilistic Model Checking on General Purpose Graphics Processors. STTT **13**(1), 21–35 (2011). https://doi.org/10.1007/s10009-010-0176-4
14. 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)
15. Bošnački, D., van den Brand, M., Denissen, P., Huizing, C., Jacobs, B., Kuiper, R., Wijs, A., Wilkowski, M., Zhang, D.: Dependency Safety for Java: Implementing Failboxes. In: PPPJ: Virtual Machines, Languages, and Tools. pp. 15:1–15:6. ACM (2016)
16. 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. pp. 141–160 (2015)
17. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.: GPU-PRISM: An Extension of PRISM for General Purpose Graphics Processing Units. In: PDMC. pp. 17–19. IEEE (2010). https://doi.org/10.1109/PDMC-HiBi.2010.11
18. Büchi, J.: On a decision method in restricted second order arithmetic. In: CLMPS, pp. 425–435. Stanford University Press (1962)
19. 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)
20. Cassee, N., Neele, T., Wijs, A.: On the Scalability of the GPUexplore Explicit-State Model Checker. In: GaM. EPTCS, vol. 263, pp. 38–52. Open Publishing Association (2017)
21. Chaki, R., Wijs, A.: Formally Characterizing the Effect of Model Transformations on System Properties. In: FACS. LNCS, vol. 13712, pp. 39–58. Springer (2022)
22. Di Ruscio, D., Kolovos, D., de Lara, J., Pierantonio, A., Tisi, M., Wimmer, M.: Low-code development and model-driven engineering: Two sides of the same coin? Software and Systems Modeling **21**, 437–446 (2022)
23. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L.: Spot 2.0—a framework for ltl and ω-automata manipulation. In: ATVA. LNCS, vol. 9938, pp. 122–129. Springer (2016)
24. Ehrig, H., Pfender, M., Schneider, H.: Graph-grammars: An algebraic approach. In: SWAT. pp. 167–180. IEEE Computer Society Press (1973)
25. Engelen, L.: From Napkin Sketches to Reliable Software. Ph.D. thesis, Eindhoven University of Technology (2012)
26. Feijen, W., van Gasteren, A.: The Alternating Bit Protocol. In: On a Method of Multiprogramming, chap. 30, pp. 333–345. Monographs in Computer Science, Springer (1999)
27. Giannakopoulou, D.: Model Checking for Concurrent Software Architectures. Ph.D. thesis, University of London (1999)

28. Giese, H., Lambers, L.: Towards Automatic Verification of Behavior Preservation for Model Transformation via Invariant Checking. In: Proc. 6th International Conference on Graph Transformation (ICGT 2012). LNCS, vol. 7562, pp. 249–263. Springer (2012)
29. Glabbeek, R.v., Luttik, S., Trčka, N.: Branching bisimilarity with explicit divergence. Fundam. Inf. **93**(4), 371–392 (Dec 2009)
30. Haak, L., Wijs, A., van den Brand, M., Huisman, M.: Formal Methods for GPGPU Programming: Is The Demand Met? In: iFM. LNCS, vol. 12546, pp. 160–177. Springer (2020)
31. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing **6**(5), 512–535 (1994)
32. Holzmann, G.: The Model Checker Spin. IEEE Trans. Software Eng. **23**(5), 279–295 (1997). https://doi.org/10.1109/32.588521
33. Hülsbusch, M., König, B., Rensink, A., Semenyak, M., Soltenborn, C., Wehrheim, H.: Showing Full Semantics Preservation in Model Transformations - A Comparison of Techniques. In: Proc. 8th International Conference on Integrated Formal Methods (iFM 2010). LNCS, vol. 6396, pp. 183–198. Springer (2010)
34. 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)
35. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Satellite Events at the MoDELS 2005 Conference. LNCS, vol. 3844, pp. 128–138. Springer (2005)
36. Kant, G., Laarman, A., Meijer, J., Pol, J.v., Blom, S., Dijk, T.: LTSmin: High-Performance Language-Independent Model Checking. In: TACAS. LNCS, vol. 9035, pp. 692–707. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_61
37. Khan, M., Hassan, O., Khan, S.: Accelerating SpMV Multiplication in Probabilistic Model Checkers Using GPUs. In: ICTAC. LNCS, vol. 12819, pp. 86–104. Springer (2021). https://doi.org/10.1007/978-3-030-85315-0_6
38. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture(TM): Practice and Promise. Addison-Wesley Professional (2005)
39. Kozen, D.: Results on the Propositional μ-Calculus. Theor. Comput. Sc. **27**(3), 333–354 (1983)
40. 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)
41. Lang, F.: Exp.Open 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-The-Fly Verification Methods. In: IFM'05. LNCS, vol. 3771, pp. 70–88. Springer (2005)
42. Leiserson, C.E., Thompson, N.C., Emer, J.S., Kuszmaul, B.C., Lampson, B.W., Sanchez, D., Schardl, T.B.: There's plenty of room at the top: What will drive computer performance after Moore's law? Science **368**(6495) (2020). https://doi.org/10.1126/science.aam9744
43. Leroy, X.: Formal proofs of code generation and verification tools. In: SEFM. LNCS, vol. 8702, pp. 1–4. Springer (2014)
44. Linden, A., Wolper, P.: A verification-based approach to memory fence insertion in PSO memory systems. In: TACAS. LNCS, vol. 7795, pp. 339–353. Springer (2013)
45. Mateescu, R., Wijs, A.: Property-Dependent Reductions Adequate with Divergence-Sensitive Branching Bisimilarity. Science of Computer Programming **96**(3), 354–376 (2014)

46. Narayanan, A., Karsai, G.: Towards Verifying Model Transformations. In: Proc. 7th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2008). ENTCS, vol. 211, pp. 191–200. Elsevier (2008)
47. Neele, T., Wijs, A., Bošnački, D., van de Pol, J.: Partial Order Reduction for GPU Model Checking. In: ATVA. LNCS, vol. 9938, pp. 357–374. Springer (2016). https://doi.org/10.1007/978-3-319-46520-3_23
48. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: SPIN 2007. LNCS, vol. 4595, pp. 263–267 (2007). https://doi.org/10.1007/978-3-540-73370-6_17
49. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (FOCS), pp. 46–57. IEEE Computer Society (1977)
50. Putter, S.d., Wijs, A.: Verifying a Verifier: On the Formal Correctness of an LTS Transformation Verification Technique. In: Proc. 19th International Conference on Fundamental Approaches to Software Engineering (FASE 2016). LNCS, vol. 9633, pp. 383–400. Springer (2016), http://dx.doi.org/10.1007/978-3-662-49665-7_23
51. Putter, S.d., Wijs, A.: A Formal Verification Technique for Behavioural Model-To-Model Transformations. Formal Aspects of Computing **30**(1), 3–43 (2018), https://link.springer.com/article/10.1007/s00165-017-0437-z
52. de Putter, S., Wijs, A.: Lock and Fence When Needed: State Space Exploration + Static Analysis = Improved Fence and Lock Insertion. In: iFM. LNCS, vol. 12546, pp. 297–317. Springer (2020)
53. 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). https://doi.org/10.1007/978-3-030-02146-7_15
54. Ragan-Kelley, J., Adams, A., Sharlet, D., Barnes, C., Paris, S., Levoy, M., Amarasinghe, S., Durand, F.: Halide: Decoupling algorithms from schedules for high-performance image processing. Commun. ACM **61**(1), 106–115 (Dec 2017). https://doi.org/10.1145/3150211
55. Rahim, L., Whittle, J.: A Survey of Approaches for Verifying Model Transformations. Software and Systems Modeling pp. 1–26 (2013). https://doi.org/10.1007/s10270-013-0358-0, http://dx.doi.org/10.1007/s10270-013-0358-0
56. Sakar, Ö., Safari, M., Huisman, M., Wijs, A.: Alpinist: An Annotation-Aware GPU Program Optimizer. In: TACAS, LNCS, vol. 13244, pp. 332–352. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99527-0_18
57. Shasha, D., Snir, M.: Efficient and correct execution of parallel programs that share memory. ACM Transactions on Programming Languages and Systems **10**(2), 282–312 (1988)
58. Stenzel, K., Moebius, M., Reif, W.: Formal Verification of QVT Transformations for Code Generation. In: MODELS. LNCS, vol. 6981, pp. 533–547. Springer (2011)
59. van der Vegt, S., Laarman, A.: A Parallel Compact Hash Table. In: MEMICS. LNCS, vol. 7119, pp. 191–204. Springer (oct 2011). https://doi.org/10.1007/978-3-642-25929-6_18
60. Wijs, A.J.: Define, Verify, Refine: Correct Composition and Transformation of Concurrent System Semantics. In: Proc. 10th International Symposium on Formal Aspects of Component Software (FACS 2013). LNCS, vol. 8348, pp. 348–368. Springer (2013)
61. Wijs, A.J., Engelen, L.J.P.: Efficient Property Preservation Checking of Model Refinements. In: Proc. 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013). LNCS, vol. 7795, pp. 565–579. Springer (2013)

62. Wijs, A.J., Engelen, L.J.P.: Refiner: Towards Formal Verification of Model Transformations. In: Proc. 6th NASA Formal Methods Symposium (NFM 2014). LNCS, vol. 8430, pp. 258–263. Springer (2014)
63. Wijs, A., Bošnački, D.: Improving GPU Sparse Matrix-Vector Multiplication for Probabilistic Model Checking. In: SPIN. LNCS, vol. 7385, pp. 98–116. Springer (2012). https://doi.org/10.1007/978-3-642-31759-0_9
64. Wijs, A., Engelen, L.: Incremental Formal Verification for Model Refining. In: MoDeVVa. pp. 29–34. ACM Press (2012)
65. Wijs, A., Osama, M.: GPUexplore 3.0: GPU Accelerated State Space Exploration for Concurrent Systems with Data. In: SPIN. LNCS, vol. 13872, pp. 188–197. Springer (2023)
66. Wijs, A., Osama, M.: A GPU Tree Database for Many-Core Explicit State Space Exploration. In: TACAS. LNCS, Springer, to appear (2023)
67. Wijs, A., Wiłkowski, M.: Modular Indirect Push-Button Formal Verification of Multi-threaded Code Generators. In: SEFM. LNCS, vol. 11724, pp. 410–429. Springer (2019)
68. Wijs, A.: BFS-Based Model Checking of Linear-Time Properties With An Application on GPUs. In: CAV, Part II. LNCS, vol. 9780, pp. 472–493. Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_26
69. Wijs, A., Bošnački, D.: GPUexplore: Many-Core On-the-Fly State Space Exploration Using GPUs. In: TACAS. LNCS, vol. 8413, pp. 233–247 (2014). https://doi.org/10.1007/978-3-642-54862-8_16
70. Wijs, A., Bošnački, D.: Many-Core On-The-Fly Model Checking of Safety Properties Using GPUs. STTT **18**(2), 169–185 (2016). https://doi.org/10.1007/s10009-015-0379-9
71. 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). https://doi.org/10.1007/978-3-319-48989-6_42
72. Zhang, D., Bošnački, D., van den Brand, M., Engelen, L., Huizing, C., Kuiper, R., Wijs, A.: Towards Verified Java Code Generation from Concurrent State Machines. In: AMT@MoDELS. pp. 64–69 (2014)
73. Zhang, D., Bošnački, D., van den Brand, M., Huizing, C., Jacobs, B., 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)