The SLCO Framework for Verified, Model-Driven Construction of Component Software

Sander de Putter[1(✉)], Anton Wijs[1], and Dan Zhang[1,2]

[1] Eindhoven University of Technology, Eindhoven, Netherlands
{s.m.j.d.putter,a.j.wijs,d.zhang}@tue.nl
[2] University of Twente, Enschede, Netherlands
d.zhang-3@utwente.nl

Abstract. We present the Simple Language of Communicating Objects (SLCO) framework, which has resulted from our research on applying formal methods for correct and efficient model-driven development of multi-component software. At the core is a domain specific language called SLCO that specifies software behaviour. In this paper, we discuss the language, give an overview of the features of the framework, and discuss our roadmap for the future.

1 Introduction

The development of complex, multi-component software is time-consuming and error-prone. One important cause is that there are multiple concerns to address. In particular, the software should be functionally correct, but also efficient. Careless optimisation of code may introduce bugs and make it less obvious to reason about the core functionality. To improve this, it is crucial that techniques are developed that make every step in the development work flow systematic and transparent.

With the Simple Language of Communicating Objects (SLCO) framework, we conduct research on the development of techniques for this purpose. Key characteristics are (1) the use of a Domain-Specific Language (DSL) based on well-known software engineering concepts, i.e., objects, variables, state machines, and sequences of instructions, (2) formal verification in every development step, from model to code, that does not require expert verification knowledge from the developer, and (3) (optimised) code generation, by which (parallel) programming challenges are hidden from the developer.

The framework uses the *model-driven software development* methodology, in which models are constructed and transformed to other models and code by means of *model transformations*. The framework makes use of a verified code

Sander de Putter—This work is supported by ARTEMIS Joint Undertaking project EMC2 (grant nr. 621429).
Dan Zhang—This work is supported by the 4TU-NIRICT project ViGARO.

generator [27,28]. Furthermore, the framework supports some verification of model-transformations; support will be extended in the near future.

2 Related Work

In most related work, no verification is done (e.g. [9,20]) or only on either model-to-model or model-to-code transformations [19]. Some techniques cover both, e.g. [14], but they do not address the *direct* verification of transformations. This means that correctness of a transformation cannot be determined once-and-for-all; instead, every time it is applied, its result has to be checked. Furthermore, a few transformation steps may quickly render verification infeasible [2]. In contrast, the SLCO framework supports direct verification of transformations. We have yet to achieve direct verification of *all* transformations, but transformations of transition sequences consisting of user-defined actions can already be verified.

SCADE 6 [10], SIMULINK [21], and EVENT-B [1] are frameworks offering features similar to SLCO. All frameworks offer verification methods for their models and automatic code generation. SCADE can make use of Lustre's verified compiler [5] to generated code. Both SIMULINK and EVENT-B support verification of generated code [11,15], however, to our knowledge the generators are not mechanically verified and, thus, require some form of consistency verification between model and code.

Unlike SCADE, SLCO is not limited to the sampling-actuating model of control engineering, and can be used to specify such systems via user-defined actions serving as sampling and actuating calls. Of the frameworks mentioned above, only EVENT-B offers verification of refinement transformations. SLCO, in addition, also supports verification of other kinds of transformations. Finally, similar to SCADE, the SLCO code generator is mechanically verified and preserves certain correctness criteria, such as atomicity preservation and lock-deadlock freedom, without the need for a consistency check.

3 The SLCO 2.0 Language

The second version of SLCO is the core of the framework. The SLCO DSL should be used in the first development step to specify the intended functionality of the system. SLCO has been designed to model systems consisting of concurrent, communicating components at a convenient level of abstraction. It has a formal semantics.[1] New to version 2 is the support for arrays, user-defined actions, composite statements, the specification of a channel's buffer capacity, and transition priorities. We will introduce these additions together with the rest of SLCO.

SLCO models consist of a finite number of *classes*, which can be instantiated as *objects*, *channels* for communication between objects, and user-defined *actions*; each are declared in their own section of the model.

[1] See http://www.win.tue.nl/~awijs/SLCO/SLCO2doc.pdf.

A *class* consists of a finite number of concurrent *state machines*, and *ports* and *variables* shared by them that can be used for communication. Thus, SLCO supports components at two levels: each object forms a component that can communicate with other objects via message-passing, while inside an object multiple components may exist that can interact via shared variables.

Variables are of type `Integer`, `Byte`, `Boolean`, or Array of one of these. Furthermore, state machines can have private variables that are only accessible by the owning state machine. Variables are declared in the `variables` section of classes or state machines: `Integer x := y + 1` declares an integer variable named `x` that initially has the value of the expression `y + 1`.

A *channel* connects two ports of two objects; they are used to send messages between state machines of two different objects. A channel accepts messages (optionally with parameters of types `Integer`, `Byte`, or `Boolean`), it is either synchronous or asynchronous and in the latter case either lossless or lossy (lossy means that it may lose messages at any time). In case a channel is asynchronous, a buffer size can be defined, which is by default 1. Let `p` and `q` be objects with ports `InOut`, `In`, and `Out`; `c(Byte) sync between p.InOut and q.InOut` and `c(Byte) async[2] lossy from p.Out to q.In` respectively denote a synchronous and a lossy asynchronous channel named `c` that accepts messages with one `Byte` parameter. The asynchronous channel may buffer up to two messages. A *port* is attached to at most one channel. Furthermore, messages sent over ports have a *name* and optionally a number of parameters with the same types as defined on the connected channel.

A state machine consists of local variables, a finite number of states, an initial state, and transitions between states. Transitions have an optional priority and a (possibly empty) sequence of *statements* associated with it; for instance, given user-defined action `a` and variable `x`, `1: s1 -> s2 {x := x + 1; a}` denotes a transition, with the priority 1, starting at state `s1` that first performs the statement `x := x + 1` and then performs the action `a`. Upon completion of the statements the state `s2` is reached. A lower number indicates higher priority. Higher priority transitions are considered for firing before lower priority ones. Transitions with the same priority are fired non-deterministically. By default, a transition has priority 0.

Parallel execution of transitions is formalised using an interleaving semantics, in which SLCO statements are atomic, i.e., the transition with sequence of statements is equivalent to a sequence of transitions each executing one of the statements in the same order. No finer-grained interleaving is allowed. SLCO offers five types of statements, some of which may sometimes be blocked from execution. When this is the case, then the associated transition is blocked as well. The five types of statements are:

1. *(Boolean) Expression*: a condition that is blocked if it evaluates to `false`. In an expression, state machine-local and object-local variables may be referenced.
2. *Assignment*: `x := e` indicates that the evaluation of an expression `e` is assigned to variable `x`. The expression may be a logical (boolean) or arith-

```
1  model Test {
2    actions init
3    classes
4      P { ... }
5      Q {
6        variables Integer x y
7        ports Out1 Out2 InOut
8        state machines
9        SM1 {
10         variables Boolean started:=false
11         initial Com0 states Com1 Com2
12         transitions
13           Com0 -> Com1 { send M(false,0) to Out1;
14                          started:=true }
15           Com1 -> Com1 { [x > 0; x:=x-1; y:=y+1];
16                          send N(y) to Out2}
17         1: Com1 -> Com2 { receive S() from InOut }
18           Com2 -> Com0 { init }
19       }
20       SM2 { ... }
21     }
22   objects p: P(), q: Q(x:=10, y:=0)
23   channels
24     c1(Boolean, Integer) async[2] lossless
25       from q.Out1 to p.In1
26     c2(Integer) async lossy from q.Out2 to p.In2
27     c3() sync between p.InOut and q.InOut
28 }
```

```
  ...
case Com1:
  // [x > 0; x := x - 1; y := y + 1]
  java_lockIDs[0] = 0; java_lockIDs[1] = 1;
  java_kp.lock(java_lockIDs,2);
  if (!(x > 0)) {
    // receive S() from InOut
    SignalMessage m = c3.receive("S");
    java_kp.unlock(java_lockIDs,2);
    if (!(m == null)) {
      // Change state
      java_currentState =
        Test.java_State.Com2;
    }
    break;
  }
  x = x - 1; y = y + 1;
  java_kp.unlock(java_lockIDs,2);
  // send N(y) to Out2
  java_lockIDs[0] = 1;
  java_kp.lock(java_lockIDs,1);
  c2.blocked_send("N", y);
  java_kp.unlock(java_lockIDs,1);
  // Change state
  java_currentState = Test.java_State.Com1;
  break;
case Com2:
  ...
```

Fig. 1. An example SLCO model (left) and part of its Java implementation (right)

metic expression. Again, both state machine-local and object-local variables may be referenced. An assignment is always able to fire.

3. *Composite*: a statement grouping an optional boolean expression and one or more assignments (in that order). It is enabled iff the expression at the head is enabled. If no expression is included, it is always able to fire. For instance, `[x>0; x:=x-1; y:=y+1]` (the square brackets denote a composite statement) indicates that in case `x` is greater than 0, `x` is decremented and `y` is incremented, all in one atomic step.
4. *Send* and *Receive*: these statements attempt to send or receive a message to or from a particular channel, respectively. If the buffer associated to the channel is full, a send operation is blocked. A receive operation is blocked if the buffer is empty, or if the next message is not as expected; the receive statement can store the received parameter values in variables, and check whether an expression related to these values evaluates to `true`. If not, the receive statement is not enabled, and communication fails.
5. *User-defined action*: an action that indicates yet-unspecified behaviour. User-defined actions can be implemented in code or transformed to concrete behaviour.

Figure 1 presents part of an SLCO model `Test` on the left. It defines classes `P`, `Q` (lines 4, 5). At line 6–7 variables `x` and `y` and ports `Out1`, `Out2` and `InOut` are defined. Class `Q` contains state machines `SM1`, `SM2` (lines 9–20). At line 22, objects `p` and `q` are declared as instances of `P` and `Q`, respectively. The object ports are attached to channels at lines 24–27. Channel `c1` accepts messages with a `Boolean` and an `Integer` parameter.

State machine `SM1` has a boolean local variable `started` (line 10), an initial state `Com0`, and other states `Com1` and `Com2` (line 11).

Between the states, transitions with sequences of atomic statements are defined (lines 12–18). The priority of the transition at line 17 enforces `SM1` in

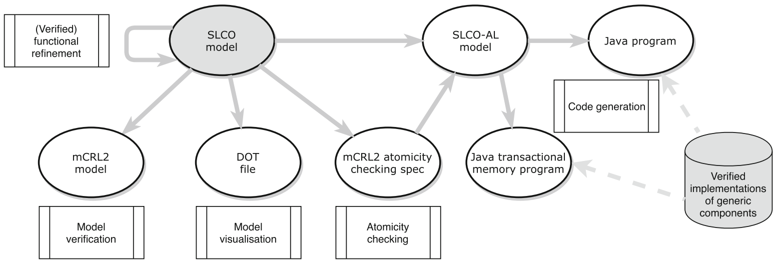
(Verified) functional refinement
SLCO model
SLCO-AL model
Java program
Code generation
mCRL2 model
DOT file
mCRL2 atomicity checking spec
Java transactional memory program
Verified implementations of generic components
Model verification
Model visualisation
Atomicity checking

Fig. 2. An overview of the SLCO framework

state `Com1` to consider the `Com1` self-loop (line 15) before the transition to `Com2` at line 17, i.e., it is first checked whether `x > 0`, only if this is not the case the transition to `Com2` is considered. A message named `M` with parameters `false` and `0` is sent at line 13, while the transition at line 17 attempts to receive a message named `S` without parameters. At line 18, a user-defined action `init` is used, to indicate that some unspecified initialisation procedure is to be performed when moving from `Com2` to `Com0`.

4 Features of the Framework

Figure 2 provides an overview of the SLCO framework. The framework[2] is implemented in Python, using TEXTX [8] for meta-modelling and JINJA2[3] for model transformation. Given an SLCO model (top-left corner), a number of features can be used.

Formal verification of SLCO *models.* To formally verify that an SLCO model satisfies desirable functional properties, it can be transformed to an MCRL2 model. With the MCRL2 toolset [7], it is then possible to apply model checking [3]. Properties specified as μ-calculus formulas can be checked by first combining model and property into a Parameterised Boolean Equation System, and then checking the latter's state space.

Model visualisation. SLCO models can be transformed to DOT files to visualise the state machines, thereby providing more insight into the structure of a model.

SLCO *model-to-model transformations.* Transformations can be used to iteratively refactor or refine SLCO models, for instance rewrite state machines or replace user-defined actions with concrete behaviour. Some user-defined transformations, specifically the ones between patterns of user-defined actions, can be verified directly for the preservation of functional properties, using our transformation verification technique [16,22] implemented in the REFINER tool [23]. It checks whether a transformation introduces patterns that are branching bisimilar to the replaced patterns after abstraction w.r.t. a given property. In other

[2] Git repository of the framework: https://gitlab.tue.nl/SLCO/SLCO.git.
[3] http://jinja.pocoo.org.

cases, preservation of properties can be determined for specific transformation applications by verifying the resulting SLCO model via MCRL2.

Transformation to multi-threaded Java. Before code is generated an SLCO model is translated to an SLCO-AL (SLCO Annotated Level) model. An SLCO-AL model is an SLCO model that is more specific on how and where to ensure atomicity.

The SLCO framework offers two partly verified code generators that take an SLCO-AL model and generate multi-threaded Java code. The generators use different methods to ensure atomicity of statements: the first generator uses a locking mechanism, while the second generator uses transactional memory.

In both generators, each state machine in the given SLCO-AL model is mapped to an individual thread. Hence, any variables shared by state machines correspond with shared variables in the Java code. The code is constructed modularly: implementations of generic concepts that are reusable in the generated code, such as channel and a locking mechanism for shared variables, have been added to a *generic component library* [27]. We have proven functional correctness of these parts of the generator [6] using VERIFAST [13]: (1) the atomicity of statements is preserved in generated code, (2) messages sent over lossless channels are eventually received, and (3) generated code does not introduce deadlocks. In addition, we have verified a robustness mechanism called *Failbox* [27] that is applied in the code to ensure that in case of a malfunctioning thread, dependent threads are notified if a thread fails.

The *first generator* enforces the use of a nested locking mechanism [28] to ensure that variables are safely shared. Each variable (and each array cell) is associated with an individual lock, and whenever for the execution of a statement a number of shared variables needs to be accessed, it is attempted to acquire the corresponding locks in a predefined order. The use of the fixed order prevents deadlocks and we have proven that it ensures the preservation of the atomicity of SLCO statements [28].

On the right in Fig. 1, part of the Java implementation of model `Test` produced by the first generator is presented. This part covers the transitions at lines 15–17 in the SLCO model and is part of a `switch` construct inside a `while` loop. This loop is responsible for the continuous movement between state machine states.

In the SLCO composite construct at line 15 in the SLCO model, class variables `x` and `y` are accessed. For this reason, locks need to be acquired in the generated code for both variables before the statement can be executed. At line 4 in the code, the IDs for both variables are added to array `java_lockIDs` in a sorted way to ensure ordered locking. Next, the locks are requested (line 5). If the locks are granted and the guard expression evaluates to `true`, the assignments of the composite statement are executed (line 17). Note the releasing of the locks once a statement has been executed. Alternatively, if the locks were not acquired or the guard expression evaluated to `false`, it is attempted to perform the `receive` statement specified at line 17 of the SLCO model. If the `receive` statement succeeded (line 10), the code ‘changes state’ according to the SLCO transition description (lines 12–13). Finally, at lines 19–23, the `send` statement at line 16 of the Slco model is executed. It is executed as a possibly blocking send operation, since in the model, the state machine is in the middle of executing the statements of the transition at lines 15–16, and cannot consider alternatives.

The *second generator* enforces the use of transactional memory, relying on the AtomJava code translation [12]. Instead of our nested locking mechanism, `atomic` blocks are used, to indicate that whenever the execution of a statement accesses a variable simultaneously accessed by another thread, the execution should be rolled back.

Reducing the use of synchronisation constructs with Slco-al**.** Naively using nested locking or `atomic` blocks for *all* statements often leads to congestion and, thus, results in under-performing parallel programs. As previously mentioned, Slco-al can be used to instruct code generators. It extends Slco with constructs to indicate synchronisation.

Some statements do not actually need protection by a synchronisation mechanism; in such a case, the lack of such synchronisations is not *observable*. Furthermore, it is possible that statements within a composite statements can be factored out in a way that the model remains *observably* equivalent. To detect such situations, the framework provides a transformation to mCRL2 that encodes an atomicity detection and avoidance algorithm based on work on atomicity checking of parallel programs [18]. This algorithm checks which specified data accesses in a model need to be protected in the code by a synchronisation mechanism in order to avoid potential *atomicity violations*. Furthermore, the algorithm determines when a fence suffices as an alternative to the more heavy-weight locks or atomic blocks. With the output of the algorithm, composite statements can be decomposed and the need for synchronisations can be indicated in an Slco-al model. In the adapted model the use of these synchronisations is restricted to the absolutely necessary and least costly ones. The adapted model is semantically indistinguishable from the original one during execution of their respective generated code.

5 Roadmap

In the near future, we will continue our research in a number of directions. For instance, it is our goal that most, if not all, Slco model transformations will be directly verifiable. Our current technique in Refiner [16,23] is restricted to transformations between action patterns, as opposed to the transformation of (patterns of) other types of Slco statements. Establishing that a transformation preserves properties for arbitrary input is stronger than having to verify resulting models each time the transformation is applied.

Regarding model verification, we plan to work on a new version of our GPU accelerated model checker GPUexplore [26] that accepts Slco models as input. Great speedups over 500× have been reported with this tool, and connecting Slco will make it feasible to rapidly produce verification results for larger models. We will also continue our research on compositional model checking [17], to modularly verify Slco models.

Regarding the SLCO-AL language, we will consider extending it to cover various other optimisation possibilities. In that respect, one can also think of optimising code w.r.t. other criteria than performance, such as power efficiency and security. To make smart decisions regarding quantitative characteristics of models, it may be required to extend our analysis towards probabilistic or stochastic model checking [3], and to add support for modelling quantitative aspects [24,25].

Research on verified code generation will focus on verifying complete programs, as opposed to only verifying generic components. We plan to use VerCors [4] for this.

We plan to address the development of GPU software. For this, we need to extend SLCO to model such systems, and construct additional code generators.

Finally, we are considering to integrate our tool chain into the ECLIPSE IDE, to create one environment in which all tools in the framework can be accessed.

References

1. Abrial, J.R., Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (2005)
2. Andova, S., van den Brand, M.G.J., Engelen, L.: Reusable and correct endogenous model transformations. In: Hu, Z., de Lara, J. (eds.) ICMT 2012. LNCS, vol. 7307, pp. 72–88. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30476-7_5
3. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
4. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The vercors tool set: verification of parallel and concurrent software. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 102–110. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_7
5. Bourke, T., Brun, L., Dagand, P.E., Leroy, X., Pouzet, M., Rieg, L.: A formally verified compiler for lustre. In: PLDI, pp. 586–601. ACM SIGPLAN Notices. ACM, New York (2017)
6. Bošnački, D., et al.: Towards modular verification of threaded concurrent executable code generated from DSL models. In: Braga, C., Ölveczky, P.C. (eds.) FACS 2015. LNCS, vol. 9539, pp. 141–160. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-28934-2_8
7. Cranen, S., et al.: An overview of the mCRL2 toolset and its recent advances. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 199–213. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_15
8. Dejanović, I., Vaderna, R., Milosavljević, G., Vuković, Ž.: TextX: a python tool for Domain-Specific Languages implementation. Knowl.-Based Syst. **115**, 1–4 (2017)
9. Deligiannis, P., Donaldson, A., Ketema, J., Lal, A., Thomson, P.: Asynchronous programming, analysis and testing with state machines. In: PLD, vol. 50, pp. 154–164. ACM SIGPLAN Notices. ACM Press (2015)
10. Dormoy, F.X.: Scade 6: a model based solution for safety critical software development. In: ERTS, pp. 1–9 (2008)
11. Fürst, A., Hoang, T.S., Basin, D., Desai, K., Sato, N., Miyazaki, K.: Code generation for event-B. In: Albert, E., Sekerinski, E. (eds.) IFM 2014. LNCS, vol. 8739, pp. 323–338. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10181-1_20

12. Hindman, B., Grossman, D.: Atomicity via source-to-source translation. In: MSPC, pp. 82–91. ACM Press (2006)
13. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_4
14. Narayanan, A., Karsai, G.: Towards verifying model transformations. In: GT-VMT. ENTCS, vol. 211, pp. 191–200. Elsevier (2008)
15. O'Halloran, C.: Automated verification of code automatically generated from simulink®. Autom. Softw. Eng. **20**(2), 237–264 (2013)
16. de Putter, S., Wijs, A.: A formal verification technique for behavioural model-to-model transformations. Form. Asp. Comput. **30**(1), 3–43 (2017)
17. de Putter, S., Wijs, A.: Compositional model checking is lively. In: Proença, J., Lumpe, M. (eds.) FACS 2017. LNCS, vol. 10487, pp. 117–136. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68034-7_7
18. de Putter, S., Wijs, A., Zhang, D.: Model Driven Avoidance of Atomicity Violations under Relaxed-Memory Models (2018, Submitted)
19. Rahim, L., Whittle, J.: A survey of approaches for verifying model transformations. Software and Systems Modeling pp. 1–26 (2013)
20. Rompf, T., Odersky, M.: Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. Commun. ACM **55**(6), 121–130 (2012)
21. The MathWorks Inc., Simulink®. www.mathworks.com/products/simulink
22. Wijs, A., Engelen, L.: Efficient property preservation checking of model refinements. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 565–579. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_41
23. Wijs, A., Engelen, L.: REFINER: Towards formal verification of model transformations. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2014. LNCS, vol. 8430, pp. 258–263. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06200-6_21
24. Wijs, A.: Achieving discrete relative timing with untimed process algebra. In: ICECCS, pp. 35–44. IEEE (2007)
25. Wijs, A., Fokkink, W.: From χ_t to μCRL: combining performance and functional analysis. In: ICECCS, pp. 184–193. IEEE (2005)
26. Wijs, A., Neele, T., Bošnački, D.: GPUexplore 2.0: unleashing GPU explicit-state model checking. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 694–701. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_42
27. Zhang, D.: From Concurrent State Machines to Reliable Multi-threaded Java Code. Ph.D. thesis, Eindhoven University of Technology (2018)
28. Zhang, D., et al.: Verifying atomicity preservation and deadlock freedom of a generic shared variable mechanism used in model-to-code transformations. In: Hammoudi, S., Pires, L.F., Selic, B., Desfray, P. (eds.) MODELSWARD 2016. CCIS, vol. 692, pp. 249–273. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66302-9_13