Towards Verified Java Code Generation from Concurrent State Machines

Dan Zhang*[1], Dragan Bošnački[1], Mark van den Brand[1], Luc Engelen[1], Cornelis Huizing[1], Ruurd Kuiper[1], and Anton Wijs **[2]

[1] Eindhoven University of Technology, The Netherlands
[2] RWTH Aachen University, Germany

Abstract. We present work in progress on, verified, transformation of a modeling language based on communicating concurrent state machines, SLCO, to Java. Some concurrency related challenges, related to atomicity and non-standard fairness issues, are pointed out. We discuss solutions based on Java synchronization concepts.

1 Introduction

Model-Driven Software Engineering (MDSE) is gaining popularity as a methodology for developing software in an efficient way. In many cases the models can be verified which leads to higher quality software. In MDSE, an abstract model is made increasingly more detailed through model-to-model transformations, until the model can be transformed to source code. This allows detecting defects in the early stages of the development, which can significantly reduce production costs and improve end product quality.

One of the challenges in MDSE is maintaining correctness during the development. The correctness of model-to-model transformations is one of the research topics of our group [6–8]. In this paper the correctness of model-to-code transformations is addressed. We investigate automated Java code generation from models in the domain specific language Simple Language of Communicating Objects (SLCO) [1]. SLCO was developed as a small modeling language with a clean manageable semantics. It allows modeling complex embedded concurrent systems. SLCO models are collections of concurrent objects. The dynamics of the objects is given by state machines that can communicate via shared memory (shared variables in objects) and message passing (channels between objects).

In this paper, we present the current status of our work and how we envision its continuation. We have defined a transformation from SLCO to Java code,

* This work was sponsored by the China Scholarship Council (CSC)
** This work was done with financial support from the Netherlands Organisation for Scientific Research (NWO).

and discuss in this paper the main complications as regards specifically efficient communication via shared variables and channels. We have started proving that this transformation is correct, i.e. that the semantics of SLCO models is preserved under some assumptions. This reasoning requires tackling non-standard fairness issues involving the built-in fairness assumptions in Java.

2 SLCO and its transformation to Java

SLCO In SLCO, systems consisting of concurrent, communicating objects can be described using an intuitive graphical syntax. The objects are instances of classes, and connected by channels, over which they send and receive signals. They are connected to the channels via their ports.

SLCO offers three types of channels: synchronous, asynchronous lossless, and asynchronous lossy channels. Furthermore, each channel is suited to transfer signals of a predefined type.

The behaviour of objects is specified using state machines, such as in Figure 1. As can be seen in the figure, each transition has a source and target state, and a list of statements that are executed when the transition is fired. A transition is enabled if the first of these statements is enabled. SLCO supports a variety of statement types. For communication between objects, there are statements for sending and receiving signals. The statement **send** $T(s)$ **to** *InOut*, for instance, sends a signal named T with a single argument s via port *InOut*. Its counterpart **receive** $T(s)$ **from** *InOut* receives a signal named T from port *InOut* and stores the value of the argument in variable s. Statements such as **receive** $P([[\textbf{false}]])$ **from** *In1* offer a form of conditional signal reception. Only those signals whose argument is equal to **false** will be accepted. There is also a more general form of conditional signal reception. For example, statement **receive** $Q(m \mid m \geq 0)$ **from** *In2* only accepts those signals whose argument is at least equal to 0. Boolean expressions, such as $m == 6$, denote statements that block until the expression holds. Time is incorporated in SLCO by means of delay statements. For example, the statement **after** 5 **ms** blocks until 5 ms have passed. Assignment statements, such as $m := m + 1$, are used to assign values to variables. Variables either belong to an object or a

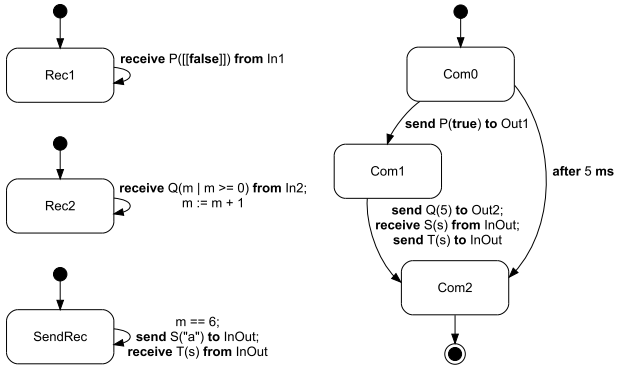

Fig. 1. Behaviour diagram of an SLCO model

state machine. The variables that belong to an object are accessible by all state machines that are part of the object.

Transformation from SLCO to Java The transformation from an SLCO model to Java is performed in two stages. First the textual SLCO model is automatically transformed into an intermediate model, which is then translated to Java. Recently we created a platform for the second stage in the Epsilon Generation Language (EGL) [2], tailored for model-to-text transformation. The execution of the generated Java code should reflect the semantics of SLCO. In the sequel we present some important parts of the end-to-end transformation from SLCO models to Java code.

State Machines Since SLCO state machines represent concurrent processes, each state machine is mapped to a different thread in the Java implementation.

We use switch statements to represent the behavior of the state machine (Listing 1.1). Each case corresponds to one state of the state machine and comprises a sequence of statements corresponding to the actions of a related outgoing state transition. If a state has more than one transition, nested switch statements are added in the generated code with a separate case for each transition.

Listing 1.1. Part of generated code for the state machine

```
1  currentState = "Com0";
2  while(true){
3    switch(currentState){
4       case "Com0":
5          String transitions[] = {"Com02Com1","Com02Com2"};
6          ...
7          int idx = new Random().nextInt(transitions.length);
8          String nextTransition = transitions[idx];
9          ...
10         switch(nextTransition){
11            case "Com02Com1":
12            ...//check whether the transition Com02Com1 is enabled or not
13         }
14      case "Com1":
15      ...
16   }
17 }
```

Shared Variables Since shared variables can be accessed and modified by multiple state machines in one object, we need to consider synchronization constructs. In our current implementation we use one writing lock and one reading lock to control the access for all shared variables in one object. Both the reading and writing locks operate in fair mode, using an approximate arrival-order policy. A boolean expression statement using a reading lock is shown in Listing 1.2.

Listing 1.2. Part of generated code for the Boolean expression statement

```
boolean isExecutedExpression = false;
while(!isExecutedExpression){
  r.lock();
  try{
    if((m.value==6)){
      isExecutedExpression = true;
    }
  } finally { r.unlock(); }
}
...
```

Channels Because lossy channels in SLCO are an undesired aspect of physical connections, these are not transformed to Java; hence the framework just supports the synchronous and asynchronous lossless channel.

SLCO's asynchronous channels have buffer capacity 1. The sender/receiver blocks when the channel is full/empty - this is provided by a BlockingQueue with a buffer capacity of 1 (from the package java.util.concurrent). In case of conditional reception from an SLCO channel, the element will only be consumed if the value satisfies the condition. This requires that the head of the queue is inspected without taking the element, which is provided by `BlockingQueue`.

SLCO's synchronous channels enable handshake-like synchronization between state machines. Our current implementation again uses the BlockingQueue, with additional ad hoc synchronization code.

In Listing 1.3 we give the generated Java code for sending a signal message of state machine Com via port InOut as shown in Fig. 1. Notice that we add the wait() method of Java to synchronize sending and receiving parties, if the channel between two state machines is synchronous.

Listing 1.3. Part of generated code of channels for sending signal message

```
synchronized(port_InOut.channel.queue){
port_InOut.channel.queue.put(new SignalMessage("T",new Object[]{s}));
if(port_InOut.channel.isSynchronousChannel){
  port_InOut.channel.queue.wait();
}...
}
```

3 Challenges

The challenges in the transformation from SLCO to Java as well as in the verification of it mainly originate from the differences between the modeling-oriented primitives in SLCO and their implementation-oriented counterparts in Java.

Shared variables – atomicity In SLCO assignments where multiple shared variables are involved are atomic, therefore we use Java variables together with locks to guard the assignments. Instead of the built-in locks we use ReentrantReadWriteLock from the package java.util.concurrent.locks with concurrent read access, which improves performance. Furthermore, in SLCO conditions on a transition may involve several shared variables. The aforementioned package enables to create a condition object for each condition on a transition, that can be used to notify waiting processes, which gives better performance than checking with busy-waiting. Our current implementation uses ReentrantReadWriteLock, but with a simple busy wait to simulate the SLCO blocking. Using the condition objects whilst maintaining the SLCO semantics is the next step.

Channels – synchronization Other synchronization primitives from the package java.util.concurrent are under investigation to replace the current ad hoc synchronization code.

Choice between transitions – external/internal In case of several possibly blocking outgoing transitions of a state in SLCO, the state machine will only block if all outgoing transitions are blocked. (This applies to shared variable access as well as channel operations.) In a naive implementation, however, the disabledness of a transition can only be assessed by initiating its execution and then blocking, and waiting for the transition becoming enabled. Essentially, this means that an external choice is turned into an internal choice, which is undesired. Here we face a similar challenge as with conditions or assignments using several shared variables. This may be solved in a similar manner, namely with a condition object dedicated to the state and cooperating processes that notify the waiting process when one of the reasons for blocking might have changed.

We have solutions for specific cases and are investigating a generic solution for all transitions that involve blocking and is orthogonal to other concurrency aspects like conditional reading from a channel, synchronization of channels, etc.

Fairness – interleaving, resolving conflict The SLCO specification approach raises non-standard fairness issues. In early usage of SLCO, no fairness was assumed for SLCO specifications. If properties depended upon fairness, this meant taking this into account for the implementation. The challenge is to advance the formal rigor of the SLCO-approach by formally expressing fairness and have the generated code together with the JVM ensure this fairness.

We use an interleaving semantics for SLCO with weak fairness: if at some time point a transition becomes continuously enabled, this transition will at some later time point be taken: in linear time temporal logic $\Box(\Box \mathit{enabled}(t) \rightarrow \Diamond \mathit{taken}(t))$.

Because the granularity in Java is much finer than in SLCO, more progress is enforced by weak fairness in SLCO than in Java. Therefore we need stronger fairness in Java. We aim to achieve this through a combination of fairness in scheduling threads, obtainable by choosing the right JVM, and fair locks, obtainable from the package java.util.concurrent.locks.

Verification We aim to partially verify the transformation. Our first approach is that the generated code uses generic code that we provide as a library of implementations for, e.g., channels. We annotate and verify this generic code. To deal with pointers and concurrency in Java we will use Separation Logic [3] in combination with the tool VeriFast [5] as we have applied to a preliminary version of this research [4]. The report indicates that this is feasible. Our research aims to incorporate more advanced library code from the java.util.concurrent packages. A second, complementary, approach deals with specific code generated in the transformation. From the SLCO model we generate annotation along with the code, which captures what the code should satisfy to conform to the SLCO semantics – whether it does is then verified using the techniques described above.

4 Conclusions

In the context of automated Java code generation we identified several challenges involving shared memory communication and fairness. Our initial findings imply that the verification of the generic Java code implementing shared memory communication for safety properties is feasible using separation logic. As one of the anonymous reviewers observed, it might be useful to identify Java patterns for correctly capturing concurrent model semantics. Besides code generation, this can be used as a basis for developing efficient simulation, formal verification and other analysis tools.

References

1. Engelen, L.: From Napkin Sketches to Reliable Software. Ph.D. thesis, Eindhoven University of Technology (2012)
2. Kolovos, D., Rose, L., Garcia-Dominguez, A., Page, R.: The Epsilon Book (2013), checked 17/07/2014
3. Reynolds, J.C.: An Overview of Separation Logic. In: Meyer, B., Woodcock, J. (eds.) VSTTE. Lecture Notes in Computer Science, vol. 4171, pp. 460–469. Springer (2005)
4. Roede, S.: Proving Correctness of Threaded Parallel Executable Code Generated from Models Described by a Domain Specific Language. Master's thesis, Eindhoven University of Technology (2012)
5. Smans, J., Jacobs, B., Piessens, F.: Verifast for Java: A tutorial. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming, Lecture Notes in Computer Science, vol. 7850, pp. 407–442. Springer (2013)
6. Wijs, A.: Define, Verify, Refine: Correct Composition and Transformation of Concurrent System Semantics. In: FACS'13. Lecture Notes in Computer Science, vol. 8348, pp. 348–368. Springer (2013)
7. 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)
8. Wijs, A., Engelen, L.: REFINER: Towards Formal Verification of Model Transformations. In: NFM'14. Lecture Notes in Computer Science, vol. 8430, pp. 258–263. Springer (2014)