Verification of Atomicity Preservation in Model-To-Code Transformations Using Generic Java Code

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

[1]*Department of Mathematics and Computer Science, Eindhoven University of Technology, Den Dolech 2, Eindhoven, The Netherlands*

[2]*Department of Computer Science, KU Leuven, Leuven, Belgium*

{*d.zhang, D.Bosnacki, M.G.J.v.d.Brand, c.huizing, R.Kuiper, A.J.Wijs*}*@tue.nl, bart.jacobs@cs.kuleuven.be*

Keywords: Model Transformation, Code Generation, Concurrency, Atomicity, Formal Verification, Separation logic.

Abstract: A challenging aspect of model-to-code transformations is to ensure that the semantic behavior of the input model is preserved in the output code. When constructing concurrent systems, this is mainly difficult due to the non-deterministic potential interaction between threads. In this paper, we consider this issue for a framework that implements a transformation chain from models expressed in the state machine based domain specific language SLCO to Java. In particular, we provide a fine-grained generic solution to preserve atomicity of SLCO statements in the Java implementation. We give its generic specification based on separation logic and verify it using the verification tool VeriFast. The solution can be regarded as a reusable module to safely implement atomic operations in concurrent systems.

1 INTRODUCTION

Model transformation is a powerful concept in model-driven software engineering (Kleppe et al., 2005). Starting with an initial model written in a domain specific language (DSL), other artifacts such as additional models, source code and test scripts can be produced via a chain of transformations. The initial model is typically written at a conveniently high level of abstraction, allowing the user to reason about complex system behaviour in an intuitive way. The model transformations are supposed to preserve the correctness of the initial model, thereby realising a framework where the generated artifacts are correct by construction. A question that naturally arises for model-to-code transformations is how to guarantee that functional properties of the input models are preserved in the generated code (Rahim and Whittle, 2013). In particular, this requires semantic conformance between the model and the generated code. For models in the area of safety-critical concurrent systems, the main complication to guarantee this equivalence involves the potential of threads to non-deterministically interact with each other.

Specifically, when variables are shared among multiple threads, the absence of race conditions is crucial to guarantee that no undesired updates of those variables can be performed. This relates to the notion of *atomicity* of the instructions executed by the threads. For instance, if two threads both increment the value of a variable x by one, then only when each of those increments can be performed atomically is it ensured that the final value of x equals the initial value plus two. Achieving atomicity of program instructions can be done using various techniques, such as locks, semaphores, mutexes, or CPU instructions such as *compare-and-swap*.

Also in modeling languages, atomicity is an important concept, to simplify the reasoning about program instructions by abstracting away the atomicity implementation details. Hence, an important requirement for model-to-code transformations is that the atomicity of the statements in the modeling language is preserved in the code. A conceptual solution would be to map each statement to an atomic block in the implementation language. Strictly speaking, a block of instructions is atomic if during its execution no instruction of another thread is allowed to be executed. However, such a definition is too strong for practical purposes, since it excludes the possibility for threads to run truly concurrently in cases when they access different variables, and therefore do not interfere with each other. For this reason, it is usually replaced with weaker notions that still ensure non-

interference. One such version, sometimes called *serializability* (Biswas et al., 2014), allows instruction blocks to be executed concurrently as long as their individual results are not affected by the other blocks.

In this paper, we demonstrate how one can establish that a model-to-code transformation transforms atomic statements in modeling languages to blocks of program instructions that are serializable. To illustrate this, we focus on a DSL called Simple Language of Communicating Objects (SLCO) (Engelen, 2012), on the one hand, and the Java programming language on the other hand. It should be stressed, though, that our approach is suitable for any combination of a modern imperative programming language with concurrency and a modeling language that is, like SLCO, based on state machines that can be placed in parallel composition, and can change state by firing transitions with atomic statements (for instance, UML state machines).

SLCO was originally introduced to model complex embedded concurrent systems by means of state machines in combination with variables. A technique to verify SLCO-to-SLCO transformations has been proposed in (Wijs, 2013; Wijs and Engelen, 2013; Wijs and Engelen, 2014). In this paper we focus on the correctness of a fully automated model-to-code transformation in which each SLCO state machine is transformed to an individual Java thread. In order to define the transformation in a modular way, and thereby improving its maintainability, we divide it into two parts, one part transforming SLCO concepts into *generic code*, and the other part transforming the aspects that are specific for the particular input SLCO model into *specific code*. The specific code may refer to the generic code to use the model-independent concepts. An example of a generic SLCO concept is the communication channel, while a particular state machine is an example of a concept specific for a given SLCO model. This way of working provides a clear maintenance advantage, as the implementations of generic concepts can be updated without affecting the overall transformation machinery. Another benefit is that the generic code needs to be verified only once. Each class of the generic code can be specified and verified in isolation, allowing for modular verification.

In the past, we have outlined the approach described above in (Zhang et al., 2014), in which we have also identified the main challenges. As a first step towards having completely verified generic code, we focussed on the SLCO communication channel, and formally verified, using the VeriFast tool (Jacobs et al., 2011), that its semantics is captured by the Java construct to which we transform it (Bošnački et al., 2015).

Contributions. Our contribution in this paper is three-fold. First of all, we discuss how we have implemented, specified, and verified a protection mechanism to access shared variables in such a way that the code blocks implementing atomic DSL statements are guaranteed to be serializable. This generic mechanism is used in our framework to automatically transform SLCO models into multi-threaded Java code, but the solution is general enough to be used in other model-to-code transformations as well.

The mechanism employs a fine-grained ordered-locking approach. A coarse-grained approach tends to negatively impact the performance of multi-threaded software, while a lock-free approach, in particular using atomic instructions such as *compare-and-swap*, is necessarily restricted to work only for statements that involve a single shared variable.

Second, we show the feasibility to verify the atomicity of generic statements by focussing on the SLCO assignment statement. We formally prove its implementation against a specification of non-interference using the VeriFast tool (Jacobs et al., 2011). Being based on separation logic (Reynolds, 2002), VeriFast is suitable to deal with aliasing and concurrency in Java, as well as with race conditions using the concept of ownership of shared resources between multi-threaded programs.

Finally, we introduce a wrapper class pattern to perform modular verification. With the wrapper class, it is possible to encapsulate data structures that are used in the code, but are not subjected to verification (for instance, because they have already been verified at an earlier stage possibly using a different tool).

The remainder of the paper is structured as follows. In Section 2 we briefly explain SLCO, the model transformation from SLCO to Java, and the essentials of separation logic and VeriFast. Section 3 describes the implementation of atomicity of SLCO statements in Java, as well as the implementation of the generic wrapper class. In Section 4, we demonstrate how to specify and verify the Java implementation. Section 5 discusses related work, and Section 6 contains our conclusions and a discussion about future work.

2 PRELIMINARIES

2.1 SLCO

In SLCO, systems consisting of concurrent, communication components can be described using an in-

tuitive graphical syntax. Objects, as instances of classes, can communicate via channels, over which they send and receive signals. Objects are connected to channels via their ports. Each object consists of a number of finite state machines and shared variables. The state machines in an object can use private, local variables, and communicate with each other via the shared variables in the object. Each transition of a state machine may have an associated SLCO *statement*. [1] SLCO offers five types of atomic statements: SendSignal, ReceiveSignal, (Boolean) Expression, Assignment, and Delay.

State machines, like the ones in Figure 1, are used to specify object behavior. Each transition has a source and target state, and the statement associated with a transition is executed when the transition is fired. Parallel execution of transitions is formalized in the form of interleaving semantics. A transition is enabled if the statement is enabled or there is no statement associated with the transition. 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** $Q(m \mid m >= 0)$ **from** *In2* offer a form of conditional signal reception. In this example, only those signals are accepted whose argument is at least equal to 0. Boolean expressions, such as m=6, denote statements that block until the expression holds. Assignment statements, such as $m := m + 1$, are always enabled, and are used to assign values to variables. Finally, time is incorporated in SLCO by means of delay statements. For example, the statement **after** 5 **ms** blocks until 5 ms have passed since the moment the source state was entered.

As already mentioned, variables can be shared by multiple state machines. In Figure 1, the three state machines in the left rectangle are part of the same object, and m is shared by them: assigning a value to m in the second state machine may affect the truth-value of the expression in the third. Statements of types Expression, Assignment, ReceiveSignal and SendSignal can all refer to variables shared by multiple state machines.

[1] There is an extended version of SLCO allowing multiple statements per transition. In this paper, we consider the basic language, since extended SLCO models can be translated to basic SLCO models (Engelen, 2012).

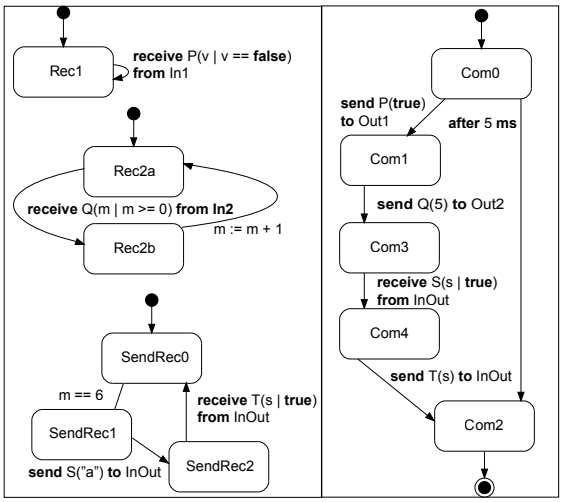

Figure 1: State machines in an SLCO model

2.2 Model Transformation

Recently, we developed an automated model-to-code transformation (AuthorsOfThisPaper, 2015) from SLCO models to multi-threaded Java programs. The transformation consists of multiple steps. Here we focus on the last step that transforms SLCO models to Java code. The preceding steps are transformations from SLCO models to more refined SLCO models using Xtend. [2] These steps are used to deal with potential semantic and platform differences.

After this problem has been resolved, the last step from SLCO to Java is applied, which is implemented in the Epsilon Generation Language (EGL) (Kolovos et al., 2011) based on Eclipse. The output is defined by means of templates that are used by the generator to produce the Java code. The generator applies transformation rules, defined in the template, to all the meta model objects which results in generation of the corresponding Java code. This Java code is constructed by combining specific code implementing the behavior of the input model with generic code implementing model-independent SLCO concepts. Examples of such concepts are the communication channel, the various types of statements, and a list datatype to store the shared variables owned by an object. The transformation achieves a one-to-one mapping between the state machines in an SLCO model and the threads in the derived program. Finally, the specific code is combined with the generic code to obtain complete, executable code that should behave as the SLCO model specifies. In order to guarantee that important properties of the input model are preserved, the transformation needs to be verified. In this paper,

[2] `www.eclipse.org/xtend`

we focus on verifying that the atomic nature of SLCO statements is preserved when they are transformed to blocks of Java instructions. The main complication when verifying this lies in the fact that the statements may access shared variables, and hence can potentially interfere with each other when executed concurrently. We use separation logic to specify the code blocks.

2.3 Separation Logic

Separation logic (Reynolds, 2002; O'Hearn et al., 2001) is an extension of Hoare logic (Owicki and Gries, 1976) that supports reasoning about shared memory which can be referenced from more than one location. Therefore, separation logic is used to describe the heap – a mapping from object IDs and object fields to values, where a value is an object or a constant. The basic heap expressions are **emp**, the empty heap, satisfied by states having a heap with no entries, and $E \mapsto F$ (read as "E points to F"), a singleton heap, satisfied by a state with a heap consisting of only one entry on address E with content F. For instance, $o.f \mapsto v$ means that field f of object o has value v. To represent complex heaps (e.g., dynamic data structures) the logical operator '$*$', called *separation conjunction* is used. Expression P*Q asserts that the heap contains disjoint parts such that P holds in one part and Q holds in the other. So, unlike its counterpart $o.f \mapsto v \wedge o.f \mapsto v$, the expression $o.f \mapsto v * o.f \mapsto v$ evaluates to **false** because the two heap components are not disjoint. In a concurrent setting, this is used to detect data races, i.e., a simultaneous access to the same memory objects by two different threads.

In addition to the standard rules of the Hoare framework, separation logic has the *frame rule*. It allows to extend the specification of a program segment C with assertion R. The axiom requires that no free variable in R is modified by C:

$$\frac{\{P\}\ C\ \{Q\}}{\{P*R\}\ C\ \{Q*R\}} \qquad \text{(frame)}$$

Separation logic uses the principle of a minimal memory footprint, meaning that a separation assertion describes a unique heap. For example, the assertion $o.f \mapsto a * o.g \mapsto b$ describes a heap consisting of exactly two entries. This property together with the requirement that the heaps of two separate threads are disjoint, makes it possible to give a natural ownership interpretation of a shared resource. If a separation logic assertion P holds at some program location on a thread, then we say that the thread owns the part of the heap described by P at that location.

The portions of the heap associated with each thread are always mutually disjoint. When a thread acquires a shared object, it claims the ownership of the state associated with the variable; when releasing the variable, it must return the ownership of the corresponding piece of state. At all stages, our use of separation logic ensures that each piece of the heap is accessed by at most one thread. It thus becomes possible to reason about concurrent programs in which ownership of a shared variable can be perceived to transfer dynamically between threads. We achieve this dynamic transfer by associating invariants to locks of shared objects. The invariant representing the environment of the thread expresses the ownership of the shared variable.

By acquiring a lock, the verified program component also acquires the lock invariant representing the heap that corresponds to the shared variables. The invariant carries a full permission to change the actual shared variables. By releasing the lock, the component releases, together with the invariant, also the acquired ownerships. This is expressed by the following rules for the *lock* and *unlock* operations

$$\{\mathbf{emp}\}\ \texttt{v.lock()}\ \{I_{\texttt{v}}(l)\} \qquad \text{(L)}$$

$$\{I_{\texttt{v}}(l)\}\ \texttt{v.unlock()}\ \{\mathbf{emp}\} \qquad \text{(UL)}$$

where $I_{\texttt{v}}(l)$ is the invariant associated with lock l of variable `v`.

To specify read-only sharing of variables fractional ownerships (permissions) are used. A fractional permission with fraction ϕ is denoted by $[\phi]o.f \mapsto v$, where $0 < \phi \leq 1$. When $\phi = 1$, the fraction is omitted and we obtain the usual $o.f \mapsto v$. This case expresses *full ownership*, allowing both read and write access.

2.4 VeriFast

The VeriFast tool is a program verifier for sequential and concurrent C and Java programs. Programs are annotated with assertions written as separation logic formulae. The verifier can check for `NULLPointerException` or `ArrayIndexOutOfBoundsException`. For concurrent programs, it checks that the program does not contain data races (memory safety). When the verification succeeds and it reports no error, the assertions and method contracts (preconditions and postconditions) are respected in every program execution. In the verification, VeriFast executes method bodies symbolically. The symbolic execution of a triple $\{P\}\ C\ \{Q\}$ starts in the symbolic state corresponding to the precondition P. If the triple is correct, each finite execution should eventually reach a symbolic state implying Q.

3 IMPLEMENTING SLCO ATOMICITY

In this section, we give the formal definition of atomicity of SLCO statements as well as a semantically comparable form of non-interference called serializability (Biswas et al., 2014) for the Java blocks implementing those statements. Furthermore, to facilitate the transformation of SLCO statements to Java code, specifically to handle accessing shared variables, we introduce the generic data structure `SharedVariableList`.

In our model-to-code transformation, each SLCO statement s in a state machine M is transformed into a block of Java instructions $\sigma = s_0; s_1; \ldots; s_n$ to be executed by a thread t_M. Strictly speaking, preserving atomicity of s in σ means that no instruction s' of some thread $t \neq t_M$ is allowed to be executed between the beginning and end of the execution of σ.

However, implementing atomicity in this strict sense is practically undesirable when constructing multi-threaded software, since it disallows true parallelism. That is why we replace this strong atomicity requirement with serializability. Serializability guarantees that for any concurrent execution of (atomic) Java blocks there exists a sequential execution of those blocks that is indistinguishable from the concurrent execution, in terms of the final effect on the global system state. More explicitly, let σ and σ' be two different instruction blocks to be executed by different threads t_M, t'_M. Let q_0 be a global state in the Java model in which both σ and σ' can start a concurrent execution and let q_1 be a state in which the system ends up after the execution of both σ and σ'. Then, q_1 also is obtained after sequential execution of the sequence $\sigma\sigma'$ or $\sigma'\sigma$ (or both). Hence, we may reason about their execution as if σ was first completely executed before σ' was started, or vice versa. (Note that this also covers the case when σ may prevent the execution of σ' or vice versa.) The extension of the concept of serializability to an arbitrary number of instruction blocks σ_i is straightforward.

The state of a system is determined by the values of its variables. In SLCO, statements may access variables shared by multiple state machines (see Section 2). Therefore, in the corresponding Java code, multiple threads may access the same shared variables. In order to realize serializability in such a setting, it must be ensured that an instruction s' of some thread t cannot affect the variables accessed by the instructions in a block σ of thread $t_M \neq t$ running concurrently.

The way in which shared variables are protected has a significant impact on the overall performance of concurrent programs. For example, using one single global lock to protect a list frequently accessed by several threads is likely to scale much worse than when each element in that list is individually lockable.

SLCO statements may require access to just a subset of the shared variables of an object. Therefore, each element in the list of shared variables is assigned its own lock for read and write access. This gives a better performance than using a single lock for the complete list. In this way we achieve serializability, as shown in Section 4.2.

Individual locking may introduce deadlocks. We use the technique of ordered locking [Havender, 1968] to prevent them. The ordered locking mechanism guarantees that when multiple threads compete over a set of variables, one thread is always able to acquire access to all of them. Of course, other threads requiring access to different shared variables are able to access these concurrently.

Note that the locks can be released in an arbitrary order. Obviously there is no deadlock during the releasing since at least one method - namely, `unlock` is active. After the locks are released we again have the situation in which multiple threads compete for the locks in fixed order.

Our synchronization mechanism for shared variables is shown in Listing 1. The `SharedVariableList`, as a wrapper class, is introduced to abstract away how the list of shared variables is implemented. It can be used to encapsulate Java data structures. The methods `lock` and `unlock` are used to acquire and release each lock of the shared variables in the list.

Listing 1: Class `Statement`

```
1  public abstract class Statement {
2  protected SharedVariableList variablesList;
3  ...
4  public void lock()
5  {
6  for (int i = 0; i < variablesList.size(); i++)
7  {
8  variablesList.get(i).lock.lock();
9  }
10 }
11 public void unlock()
12 {
13 for (int i = 0; i < variablesList.size(); i++)
14 {
15 variablesList.get(i).lock.unlock();
16 }
17 }
18 }
```

The class `Assignment` as a subclass of class `Statement` (Listing 2) contains a method called `lockAndAssign` that can be used to safely assign a new value to a shared variable. The abstract method `assign` is implemented in the subclass which is related to a concrete SLCO assignment. When a thread attempts to execute the method `assign`, it will be delayed until all locks in the `variablesList` of vari-

ables to be accessed by the `assign` method are not being used anymore by other threads.

Listing 2: Class `Assignment`

```
1 public abstract class Assignment extends Statement {
2 ...
3 public abstract void assign();
4 public void lockAndAssign() {
5 lock();
6 assign();
7 unlock();
8 }
9 }
```

As already mentioned, to store shared variables, we introduce a wrapper class `SharedVariableList`. Listing 3 shows its declaration. The concept of wrapper classes is quite common in object-oriented programming and it is a pattern in object-oriented development. The wrapper class is used to hide information of concrete Java data structures, which allows modular verification. Parts of the code that use `SharedVariableList` can be verified without involving the data structure; in fact, it may not even have been implemented yet. This helps to scale verification to larger programs, since the wrapper class needs to be analyzed only once, instead of once per call. Finally, modifying the implementation of the data structure encapsulated by the wrapper class never breaks correctness of its callers. This allows for simultaneous development and verification of code.

Listing 3: Declaration of class `SharedVariableList`

```
1 final class SharedVariableList{
2 public SharedVariableList();
3 public int size();
4 public SharedVariable get(int index);
5 boolean add(SharedVariable e);
6 }
```

4 SPECIFYING AND VERIFYING SLCO ATOMICITY

In the previous section, we explained how the atomicity of SLCO statements can be implemented using serializability. We can use separation logic in VeriFast to verify the serializability, i.e., the fact that there is non-interference between different threads.

The interpretation of correctness depends crucially on rules L and UL for the *lock* and *unlock* operations from Section 2.3. In our case in rules L and UL invariant $I_v(l)$ is of the form $v \mapsto \_$, i.e., expresses ownership of the variable v. By rule L, $I_v(l)$ is guaranteed to hold after `lock`, i.e., in the beginning of the protected code block. This means that the corresponding thread acquires the ownership of v. Similarly, at the end of the block, after `unlock`, the ownership of v is released together with invariant $I_v(l)$. Let V be the list of shared variables associated with the statement implemented in the block. By executing `lock` for each variable in V, using a combination of the L and UL rules and the frame rule from Section 2.3, an assertion I_V is established which is the conjunction of the invariants $I_v(l)$, for all $v \in V$. I_V can be seen as an invariant of the list V which expresses ownership of all variables in V by the thread. The concrete setting of our model transformation ensures that no shared variable in V is acquired or released within the protected code block. This is achieved by simply not using `lock` and `unlock` within the protected block, since these are the only statements with which one can acquire or release ownership. Together with the fact that invariant I_V holds at the beginning and at the end of the block, this implies non-interference since all variables are held exclusively by the thread during the execution of the block.

VeriFast supports modular verification in the sense that each method is verified separately. In this, each method relies on its environment to comply with the invariant. This is checked during the verification when several threads are combined. In the following sections, we specify and verify the atomicity of Java constructs corresponding with SLCO statements using separation logic via VeriFast.

In VeriFast, for each verified Java source file (.java) there is a corresponding specification file (.javaspec). The .java file contains implementations, specifications, annotations, and predicates, while the .javaspec file contains only declarations of predicates and specifications of methods with a semicolon instead of a method body. A .javaspec file can be used to verify client programs even without having the corresponding .java file that contains the definitions and the implementations mentioned in the .javaspec file. Thus, the client programs are users of the .javaspec files. For example, we only need to provide a pure (i.e., containing no implementations) specification file SharedVariableList.javaspec to VeriFast in order to verify the `Statement` class.

4.1 Class SharedVariableList Specification

Class `SharedVariableList` is specified in separation logic in a way that is in fact independent of the Java programming language. In Listing 4, the class `SharedVariableList` provides methods for modifying and querying its instances, such as `size`, `add` and `get`.

Listing 4: Specification of class `SharedVariableList`

```
1 final class SharedVariableList{
```

```
2  /*@ predicate List(list<SharedVariable> elements); @
       */
3  public SharedVariableList();
4  //@ requires true;
5  //@ ensures List(nil);
6  public int size();
7  //@ requires [?f]List(?es);
8  //@ ensures [f]List(es) &*& result == length(es);
9  public SharedVariable get(int index);
10 //@ requires [?f]List(?es) &*& 0 <= index &*& index <
       length(es);
11 //@ ensures [f]List(es) &*& result == nth(index, es);
12 boolean add(SharedVariable e);
13 //@ requires List(?es);
14 //@ ensures List(append(es, cons(e, nil))) &*& result
       ;
15 }
```

The VeriFast specific text, e.g., specifications and declarations of auxiliary variables introduced only for verification purposes, is located inside special comments delimited by `@`.

We express the state of the `sharedVariableList` instances using a mathematical list predefined in VeriFast as follows: The empty list is denoted by `nil` and a nonempty list starting by an element *h* and a tail *t* is denoted by `cons(h,t)`. In particular, predicate `List` is an abstract predicate that provides an abstract representation of the contents of the list of shared variables. More concretely, parameter `elements` is a mathematical list containing the actual program variables that are stored in the list. The actual implementation can be, for instance, a dynamic list or an array. Using the abstract predicate we can hide such implementation details during the verification. Note that `List` remains undefined in this specification stage. Its definition can be provided later together with the implementation of `SharedVariableList`.

The pre- and postconditions form the *contracts* of the methods and are denoted by the keywords `requires` and `ensures`, respectively, like in lines 4-5 in Listing 4. This contract of the constructor guarantees that an object is created corresponding to an empty list, regardless of the precondition.

The specification of `size` in lines 7 and 8 states that the method returns the length of the list. Component assertions of pre- and postconditions are separated by the spatial conjunction denoted by `&*&`. Both `[?f]` and `[f]` are fractional ownerships. The question mark `?` in front of a variable means that the matched value is bound to the variable and all later occurrences of that variable in the contract refer to this matched value. In our case the value of the fractional permission `f` in the precondition in line 7 must be the same as the one in the postcondition in line 8. Hence, the precondition in line 7 `[?f]List(?es)` expresses both that a fractional ownership with fraction `f` is required for the shared variable list corresponding with the mathematical list `es`, and records `f` and `es`. The postcondition specifies that the method returns the ownership of `es` to the caller with the same fraction `f` and the result that is returned by `size` is the length of `es`. `result` is a reserved variable name representing the return value of the method.

The precondition of `get` (line 10) requires that a valid element `index` is provided as an input parameter. The postcondition expresses that the element at position `index` in list `es` is returned using the mathematical function `nth`.

Unlike for other methods, the precondition of `add` (line 13) requires full ownership of the list `es`. As a result, the caller who owns `es` is allowed to insert an element into the list. Finally the method `add` returns the ownership of a new list that combines the list `es` with the newly inserted element `e` using the function `append`.

4.2 Class Statement Specification

The specification of class `Statement` is shown in Listing 5. The predicate constructor `lock_inv` in line 2 is essential for the preservation of serializability. It defines the lock invariant $I_v(l)$ associated with the lock of a variable `v` passed as a parameter. The assertion `v.value |->_` asserts the full ownership of *v.value*. The underscore '_' denotes an arbitrary value.

The recursive predicates `locks` and `invariants` in lines 3-7 and 8-12, respectively, are used to specify data structures without static bound on their size. The body of each predicate is a conditional assertion. If `vs` is `null` (the base case of the induction) then the value of predicate `locks` is `true` (line 5); otherwise, the inductive step asserts that the lock of the first element of the list `vs`, `head(vs)` is partially owned. This is given by `return [_]head(vs).lock |-> ?lock` in line 7, where `[_]` denotes an unspecified fraction. Besides that, invariant `lock_inv(head(vs))` is associated with the lock of the first element, via the predicate `ReentrantLock`. Predicate `ReentrantLock` is defined by VeriFast as a specification of the `ReentrantLock` class. In a similar fashion, the recursive predicate `invariants` states that a conjunction of invariants corresponding to the (locks of the) variables in list `vs` is recursively built. As mentioned above, for each variable the corresponding invariant is given by the specification `lock_inv(head(vs))`.

Predicate `Statement_lock()` is actually defined in Listing 6 and denotes that the `Statement` object is in a valid state corresponding to an abstract value given by the mathematical object list of shared variable objects `vs`. The body of method `lock` needs to establish the above mentioned invariant I_V of each variable in list `vs` which is expressed by the postcondition `invariants(vs)`. The postcondition

`invariants(vs)` is also one part of the precondition in the method `unlock`. By calling the method `unlock`, invariant I_v of each variable in list `vs` is not guaranteed to hold anymore. After that, other threads can acquire the ownership of those variables through the method `lock`.

Listing 5: Abstract Specifications of Class `Statement`

```
1  /*@
2  predicate_ctor lock_inv(SharedVariable v)(;) = v.
       value |-> _;
3  predicate locks(list<SharedVariable> vs;) =
4  vs == nil ?
5  true
6  :
7  [_]head(vs).lock |-> ?lock &*& [_]lock.ReentrantLock(
       lock_inv(head(vs))) &*& locks(tail(vs));
8  predicate invariants(list<SharedVariable> vs;) =
9  vs == nil ?
10 true
11 :
12 lock_inv(head(vs))() &*& invariants(tail(vs));
13 @*/
14 class Statement {
15 //@ predicate Statement_lock(list<SharedVariable> vs)
       ;
16 void lock();
17 //@ requires [_]Statement_lock(?vs);
18 //@ ensures invariants(vs);
19
20 void unlock();
21 //@ requires [_]Statement_lock(?vs) &*& invariants(vs
       );
22 //@ ensures true;
23 }
```

4.3 Class Statement Verification

Above we gave the formal specification of the class `Statement` in an abstract, mathematically precise and implementation-independent way. Providing specification of `SharedVariableList` is a critical factor to verify the implementation of class `Statement`. Additional predicates and annotations are also needed to verify the implementation of class `Statement`, as shown in Listing 6.

Listing 6: Verification annotations for class `Statement`

```
1  /*@
2  predicate_ctor lock_inv(SharedVariable v)(;) = ...
3  predicate locks(list<SharedVariable> vs;) = ...
4  predicate invariants(list<SharedVariable> vs;) = ...
5  @*/
6  class Statement {
7  SharedVariableList variablesList;
8  //@ predicate Statement_lock(list<SharedVariable> vs)
       = this.variablesList |-> ?a &*& a.List(vs) &*&
       locks(vs);
9  void lock()
10 //@ requires [_]Statement_lock(?vs);
11 //@ ensures invariants(vs);
12 {
13 for (int i = 0; i < variablesList.size(); i++)
14 //@ requires [_]variablesList |-> ?b &*& [_]b.List(vs
       ) &*& [_]locks(drop(i,vs)) &*& i >= 0 &*& i <=
       length(vs);
15 //@ ensures invariants(drop(old_i, vs));
16 {
17 //@ drop_n_plus_one(i,vs);
18 variablesList.get(i).lock.lock();
19 }
20 }
21 void unlock()
22 //@ requires [_]Statement_lock(?vs) &*& invariants(vs
       );
23 //@ ensures true;
24 {
25 for (int  i = 0; i < variablesList.size(); i++)
26 //@ requires [_]variablesList |-> ?b &*& [_]b.List(vs
       ) &*& [_]locks(drop(i,vs)) &*& invariants(drop(
       i,vs)) &*& i >= 0 &*& i <= length(vs);
27 //@ ensures true;
28 {
29 //@ drop_n_plus_one(i,vs);
30 variablesList.get(i).lock.unlock();
31 }
32 }
33 }
```

The part in lines 1-5 in Listing 6 is identical to lines 1-13 in the specification Listing 5. Line 8 in Listing 6 contains the definition of predicate `Statement_lock` which in Listing 5 was only specified. The part `this.variablesList |-> ?a` states that field `variableList` is defined. The last two conjuncts `a.List(vs) &*& locks(vs)` relate `variableList` with the mathematical variable `vs` of type `list` and moreover the variables in `vs` are connected to their corresponding locks. The contract of method `lock` in lines 10-11 is the same as the one in Listing 5.

In line 13 we use a `for` loop to obtain the lock of each element in the `SharedVariableList`. Besides loop invariants, VeriFast supports also loop verification by specifying a loop contract consisting of a precondition and a postcondition (Tuerk, 2009). Then the loop is verified as if it were written using a local recursive function. The contract specifies the permissions used only by a specific recursive call (i.e., corresponding to a specific value of the loop counter `i`). The precondition in line 14 matches `variablesList` with variable `b` (`[_]variablesList |-> ?b`), relates `b` to `vs` (`[_]b.List(vs)`), associates the variables from the `i`-th to the `vs.length-1`-th in list `vs` (`[_]locks(drop(i,vs))`) to their locks, and finally limits the range of the counter `i` (`i >= 0 &*& i <= length(vs)`). The segment `vs` from `i` to `vs.length-1` is obtained using the built-in function on lists `drop`. In a similar way in the postcondition in line 15, the list tail starting with `old_i` is obtained as an argument of the predicate `invariants`. Variable `old_i` refers to the value of the variable `i` at the start of the virtual function call (loop body). After the top virtual recursive call backtracks, i.e., after the loop termination, `old_i` equals 0. This implies the validity of the conjunction of all lock invariants and consequently the ownership of all variables in `vs`.

Lemma functions are used to help VeriFast to transform one assertion to another. The contract of a lemma function corresponds to a theorem, its body to the proof, and a lemma call to an application of the theorem. In our case lemma function `drop_n_plus_one(i,vs)` in line 17 tells the verifier that `drop(n,vs)` is equivalent to the concatenation of

the element `nth(vs)` with the list `drop(n+1,vs)`.

The detailed annotation of `unlock` (lines 21-32) uses the same concepts as the annotation of `lock`, therefore we do not discuss the former explicitly. It expresses that in each iteration the loop invariant shrinks instead of growing with the addition of a new conjunct, i.e., invariant associated to a lock.

The specification and annotation in the current section is sufficient to prove that predicate `invariants`, which corresponds to I_V, holds in the beginning and the end of the block implementing the statements. In Listing 2 this means that `invariants` holds immediately after `lock` in line 5 and immediately before the `unlock` in line 7. The validity of `invariants` ensures ownership of the variables in `sharedVariables`. As discussed above, by construction of our transformation (i.e., by not using `lock` and `unlock` within the protected block of the statement translation) we ensure that the block does not release this ownership. For example, in Listing 2 methods `assign` in line 6, as well as the implementations of all other types of SLCO statements, satisfies this property. VeriFast is able to verify that all relevant variables are held by the thread executing the method corresponding to the implementation of the SLCO statement. This implies serializability of the programs as it an be seen from the following arguments.

Consider two instruction blocks σ and σ' which both implement an SLCO statement. Hence, they both contain a lock protected code block. We show that they are serializable. Let V and V' be the set of variables accessed by σ and σ', respectively. Consider first the case when $V \cap V' \neq \emptyset$. Suppose that σ first acquires the ownership of all variables in V. Then σ' must wait until those variables are released. If there is some prefix σ'' of σ' which has been executed before σ acquired the variables in V, then σ' could have modified only variables which are not in V. So, this prefix could have been executed after σ terminated and therefore the sequence $\sigma\sigma'$ will produce the same variable changes, i.e., the same state as the concurrent execution of σ and σ'.

A similar argument can be used for the case $V \cap V' = \emptyset$. In this case the individual statements in σ and σ' are independent of one another and can be permuted in an arbitrary order. The set of possible sequences includes both $\sigma\sigma'$ and $\sigma'\sigma$ and they confluently lead to the same end state.

5 RELATED WORK

The detection of race condition violations in concurrent code using the lock mechanism has been addressed by a number of type-based (Farzan and Madhusudan, 2006), static (Engler and Ashcraft, 2003; Abadi et al., 2006) and dynamic analysis (Choi et al., 2002) tools. However, as shown in (Flanagan and Qadeer, 2003), a code block free of race conditions may still contain errors caused by simultaneous access to shared objects. Therefore, stronger concepts of non-interference are needed. In (Flanagan and Qadeer, 2003), a relaxed definition of atomicity was used and an atomic type system was implemented to check it. The tool DoubleChecker (Biswas et al., 2014) checks for serializability of concurrent programs based on run-time information about the dependences between threads. The above mentioned works check the correctness of programs a posteriori, i.e., after they have been fully implemented. In contrast, our approach statically verifies generic code to be used in the construction of complete programs.

There exists a substantial amount of work that deals with model-to-code transformations. For an overview, see (Rahim and Whittle, 2013). Here we focus on relevant work that deals with model-to-code transformations and uses verification based on deductive methods, like theorem proving.

In (Blech et al., 2005), a formal verification using Isabelle/HOL theorem prover is presented of a concrete algorithm that generates Java code from UML Statecharts. It is shown that the source UML model and the generated Java code are bisimilar. This is a one stage model transformation. In (Stenzel et al., 2011), a Java code generation framework is presented. The framework is based on the transformation language QVT. The theorem prover KIV is used to prove security properties and syntactic correctness. In both these works, one of the major concerns is the scalability when the transformations are applied on complex models. By splitting the transformation into producing generic and specific code, and verifying the generic concepts in isolation, we aim to have a more scalable approach.

Finally, software model checking techniques, e.g., (Chaki et al., 2003; Jhala and Majumdar, 2009), offer a supporting approach to verify code resulting from model-to-code transformations. These techniques could in particular be useful to verify the generic code. Tools like Java PathFinder (Visser et al., 2003) are natural candidates for this task. It remains to be investigated how feasible it is to apply these techniques in a modular approach like ours, in which each transformed concept is verified separately.

6 CONCLUSIONS

We have presented an approach for the verification of atomicity preservation in model-to-code transformations based on separation logic using the tool VeriFast. We applied this approach in the transformation from the domain specific language SLCO to Java.

To obtain high efficiency, we replaced the original strong atomicity requirement of SLCO with the semantically relaxed notion of serializability. The serializability was implemented by a fine-grained deadlock-free ordered locking allowing true parallelism with fine-grained granularity. We stated the serializability in terms of ownership of shared variables expressed by means of lock invariants. Using VeriFast we verified the non-interference in the Java code.

ACKNOWLEDGEMENTS

This work was done with financial support from the China Scholarship Council (CSC) and ARTEMIS Joint Undertaking project EMC2 (grant agreement 621429).

REFERENCES

Abadi, M., Flanagan, C., and Freund, S. N. (2006). Types for Safe Locking: Static Race Detection for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):207–255.

AuthorsOfThisPaper (2015). SLCOtoJava Model Transformation and Verification. `https://drive.google.com/drive/u/0/folders/0B2U1DbWZemiVRVlhbElPd0Z1NVE`.

Biswas, S., Huang, J., Sengupta, A., and Bond, M. D. (2014). DoubleChecker: Efficient Sound and Precise Atomicity Checking. In *ACM SIGPLAN Notices*, volume 49, pages 28–39. ACM.

Blech, J., Glesner, S., and Leitner, J. (2005). Formal Verification of Java Code Generation from UML Models. In *Fujaba Days*, pages 49–56.

Bošnački, D., van den Brand, M., Huizing, C., Jacobs, B., Kuiper, R., Wijs, A., and Zhang, D. (2015). Verification of Atomicity Preservation in Model-To-Code Transformations. In *FACS*, LNCS. Springer (accepted for publication).

Chaki, S., Clarke, E., Groce, A., Jha, S., and Veith, H. (2003). Modular Verification of Software Components in C. In *ICSE*, pages 385–395. IEEE.

Choi, J.-D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., and Sridharan, M. (2002). Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *ACM SIGPLAN Notices*, volume 37, pages 258–269. ACM.

Engelen, L. (2012). *From Napkin Sketches To Reliable Software*. PhD thesis, Eindhoven University of Technology.

Engler, D. and Ashcraft, K. (2003). RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 237–252. ACM.

Farzan, A. and Madhusudan, P. (2006). Causal Atomicity. In *CAV*, volume 4144 of *LNCS*, pages 315–328. Springer.

Flanagan, C. and Qadeer, S. (2003). A Type and Effect System for Atomicity. In *ACM SIGPLAN Notices*, volume 38, pages 338–349. ACM.

Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., and Piessens, F. (2011). VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NFM*, volume 6617 of *LNCS*, pages 41–55. Springer.

Jhala, R. and Majumdar, R. (2009). Software Model Checking. *ACM Computing Surveys*, 41(4):21:1–21:54.

Kleppe, A., Warmer, J., and Bast, W. (2005). *MDA Explained: The Model Driven Architecture(TM): Practice and Promise*. Addison-Wesley Professional.

Kolovos, D., Rose, L., Garca-Dominguez, A., and Paige, R. (2011). *The Epsilon Book*. Eclipse.

O'Hearn, P., Reynolds, J., and Yang, H. (2001). Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic, 15th International Workshop, CSL 2001.*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19.

Owicki, S. and Gries, D. (1976). Verifying Properties of Parallel Programs: An Axiomatic Approach. *Commun. ACM*, 19(5):279–285.

Rahim, L. and Whittle, J. (2013). A Survey of Approaches for Verifying Model Transformations. *Software & Systems Modeling (available online)*.

Reynolds, J. C. (2002). Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE.

Stenzel, K., Moebius, M., and Reif, W. (2011). Formal Verification of QVT Transformations for Code Generation. In *MODELS*, volume 6981 of *LNCS*, pages 533–547. Springer.

Tuerk, T. (2009). A Formalisation of Smallfoot in HOL. In *TPHOLs*, volume 5674 of *LNCS*, pages 469–484. Springer.

Visser, W., Havelund, K., Brat, G., Park, S., and Lerda, F. (2003). Model Checking Programs. *Automated Software Engineering*, 10(2):203–232.

Wijs, A. J. (2013). Define, Verify, Refine: Correct Composition and Transformation of Concurrent System Semantics. In *FACS*, volume 8348 of *LNCS*, pages 348–368. Springer.

Wijs, A. J. and Engelen, L. J. P. (2013). Efficient Property Preservation Checking of Model Refinements. In *TACAS*, volume 7795 of *LNCS*, pages 565–579. Springer.

Wijs, A. J. and Engelen, L. J. P. (2014). REFINER: Towards Formal Verification of Model Transformations. In *NFM*, volume 8430 of *LNCS*, pages 258–263. Springer.

Zhang, D., Bošnački, D., van den Brand, M., Engelen, L., Huizing, C., Kuiper, R., and Wijs, A. (2014). Towards Verified Java Code Generation from Concurrent State Machines. In *AMT*, volume 1277 of *CEUR Workshop Proc.*, pages 64–69. CEUR-WS.org.