Towards Modular Verification of Threaded Concurrent Executable Code Generated from DSL Models

Dragan Bošnački[1(✉)], Mark van den Brand[1], Joost Gabriels[1], Bart Jacobs[2], Ruurd Kuiper[1], Sybren Roede[1], Anton Wijs[1], and Dan Zhang[1]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
dragan@win.tue.nl
[2] KU Leuven, Leuven, Belgium

Abstract. An important problem in Model Driven Engineering is maintaining the correctness of a specification under model transformations. We consider this issue for a framework that implements the transformation chain from the modeling language SLCO to Java. In particular, we verify the generic part of the last transformation step to Java code, involving change in granularity, focusing on the implementation of SLCO communication channels. To this end we use a parameterized modular approach; we apply a novel proof schema that supports fine grained concurrency and procedure-modularity, and use the separation logic based tool VeriFast. Our results show that such tool-assisted formal verification can be a viable addition to traditional techniques, supporting object orientation, concurrency via threads, and parameterized verification.

1 Introduction

Model-Driven Software Engineering (MDSE) [18] is a methodology that recently gained popularity as a method for efficient software development. Constructing a model enables the developer to deal with difficult aspects at a higher and less complex level of abstraction. Transformations are used to create new models, source code, test scripts and other artifacts. By shifting the focus from the code to the model, MDSE allows to tackle defects in the software already in the modeling phase. Resolving errors in the early stages of the software development process reduces the costs and increases the reliability of the end product.

An important question is whether transformations are correct. Various types of correctness are relevant for model transformations, such as type correctness and correspondence correctness [22]. In earlier work, we have addressed how to determine that model-to-model transformations preserve functional properties [28–30]. In this paper, we focus on checking that model-to-code transformations preserve the behavioural semantics of the model [22]: If we have proven

R. Kuiper, A. Wijs and D. Zhang—This work was done with financial support from the China Scholarship Council (CSC), the Netherlands Organisation for Scientific Research (NWO), and ARTEMIS Joint Undertaking project EMC2 (grant agreement 621429).

C. Braga and P.C. Ölveczky (Eds.): FACS 2015, LNCS 9539, pp. 141–160, 2016.
DOI: 10.1007/978-3-319-28934-2_8

that certain functional properties hold in a model, such as the absence of data races or deadlocks, how can we ensure that those properties still hold in the generated source code?

Specifically, we focus on models of systems consisting of concurrent, interacting components, and wish to transform those models into multi-threaded software. For compositional models, an Object Oriented (OO) implementation language seems a natural choice, since it allows us to map components to objects. We have chosen Java. The modelling language we use is called SLCO (Simple Language of Communicating Objects) [1]. SLCO models consist of components that communicate through channels. Each component is described in terms of a finite number of concurrently operating state machines that can share variables. After a chain of transformations of SLCO models, in which incrementally more concrete information about the specified system can be added, multi-threaded Java code should be generated based on the last SLCO model, in which each SLCO state machine is mapped to its own thread.

SLCO has a coarse granularity that supports thinking about concurrency at a convenient high level of abstraction. On the other hand, the generated code implements concurrency through multi-threading, with a level of granularity that is much more fine-grained. This approach facilitates the development of correct, well-performing, complex software. However, the code generation step is challenging to implement, since the transition from coarse to fine-grained concurrency needs to be done in a way that correct and well-performing software can be generated.

Our approach to setting up the model-to-code transformation step is to identify the concepts in SLCO that are model independent on the one hand, and model dependent on the other. The model independent concepts can be transformed to Java once, and from that moment on referred to in all code generated from specific SLCO models. An example of a model independent SLCO concept is the communication channel, while state machines are model dependent concepts, since their structure differs from one model to another.

For the specification of the behaviour of Java objects, we opt for using separation logic [23], since it allows us to specify behaviour in a way independent of the implementation language. We require concurrency, so we actually work with the version of separation logic with fractional permissions. Full verification of semantics preservation of model-to-code transformations then involves establishing that these specifications correspond with the semantics of the corresponding SLCO constructs. For this to be possible, we require a modular approach, in which the specification of constructs can be used for the verification of code in which those constructs play a role.

As a first step, in this paper, we focus on how to formally specify the behaviour of model independent concepts, such that modular verification of code using those concepts is possible. In fact, using such specifications allows the verification of code without relying on the actual implementation of the model independent concepts, thereby truly realising a modular way of working. Our aim is to show that modular verification of model-to-code transformations of multi-component systems is necessary and feasible, and we demonstrate how this can be concretely done.

The model independent concepts are implemented in what we refer to as *generic code*. To verify this code, a theorem proving approach is called for, because the generic code contains parameters that only get concrete values when used in specific code derived from particular input models. Furthermore, since the generic code has fine-grained concurrency, we require procedure-modularity, and we use the approach from [16] that supports this. Tool-wise, we require a verification tool that supports OO code, concurrency and separation logic with fractional permissions, leading to the choice of VeriFast [15]. A procedure-modular approach can be achieved by using ghost code and abstract predicates.

Contributions. First of all, we introduce a new modular specification schema to specify the behaviour of modelling constructs in a setting where (1) fine-grained parallelism is used, and (2) the environment is general, i.e., we do not need to know anything about the environment to specify the constructs. Compared to earlier work, our schema allows a better abstraction from the implementation details of the methods being specified.

We demonstrate our approach by specifying and verifying a representative part of the generic code, namely the communication channel. This shows the feasibility of the approach, but also that judicious choices of implementation language, specification language, verification approach and tooling are required.

As mentioned in [22], proving correctness of a program is not as complex as proving correctness of a transformation that produces programs. By making a distinction between generic and specific code, the complexity of proving the correctness of model-to-code transformations can be lowered. Generic code can largely be treated as any other program, apart from the fact that it raises new concerns regarding the larger program context in which code constructs can be placed; these concerns are covered in this paper. As a result, the remaining proof obligations for the transformation as a whole can be simplified; once we turn our attention to the specific code, we can directly use the specifications of the generic code constructs. With respect to related work (Sect. 6), this is a novel way to address the correctness of model-to-code transformations.

Section 2 introduces SLCO and explains how SLCO models are transformed to Java code. Section 3 explains the essentials of separation logic. In Sect. 4, the new modular specification schema is described, and in Sect. 5 it is demonstrated how to apply the schema to specify and verify a Java implementation of the SLCO channel datatype using VeriFast. Section 6 discusses related work, and Sect. 7 contains our conclusions and pointers to future work.

2 SLCO and Its Transformation to Java

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

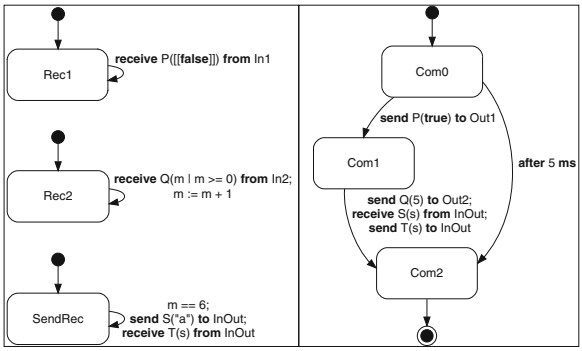

Fig. 1. Behaviour diagram of an SLCO model

The behaviour of a component is specified using a finite number of state machines, such as in Fig. 1, where two components are defined (the two main rectangles). The parallel execution of those machines is formalised in the form of interleaving semantics. Variables either belong to the whole component or an individual state machine. The variables that belong to a component are accessible by all state machines that are part of the component (for instance, variable m in the left component of Fig. 1). Each transition has a source and target state, and a list of statements that are executed when the transition is fired. A transition can be fired if it is enabled, and it is enabled if the first of the associated statements is enabled. If a transition is fired but subsequent statements are blocked, the transition blocks until they become enabled. SLCO supports a variety of statement types. For communication between components, there are statements for sending and receiving signals over channels. 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. A send statement is enabled if the buffer of the channel is not full, and a receive statement is enabled if there is a message in the buffer.

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. Another example is the statement **receive** $Q(m \mid m \geq 0)$ **from** *In2*, which only accepts those signals whose argument is greater than or equal to 0. For the above statements to be enabled, there must be a message available in the channel buffer satisfying the conditions.

Boolean expressions, such as $m{==}6$, denote statements that are enabled iff the expression holds. Time is incorporated 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. Assignment statements, such as $m := m + 1$, are used to assign values to variables. They are always enabled.

Our approach to derive executable code from an SLCO model is as shown in the activity diagram of Fig. 2: generic code constructs are used for the basic elements in SLCO, i.e., for channels (synchronous and asynchronous), states, transitions, and a mechanism to move between states by performing transitions. A model-to-code transformation takes an SLCO model as input and produces

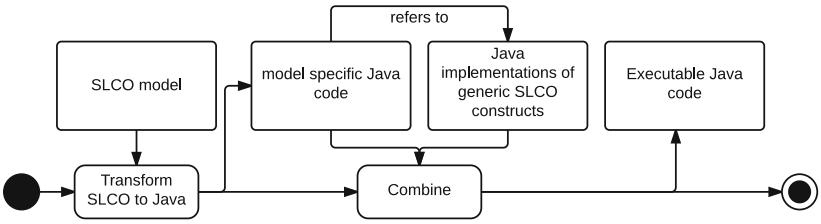

Fig. 2. Activity diagram of the transformation process from SLCO to Java

model specific Java code that refers to the generic constructs as output. There is a one-to-one mapping between SLCO state machines and Java threads. Finally, this specific code is combined with the generic code to obtain a complete, executable implementation that should behave as the SLCO model specifies.

3 Separation Logic

Separation logic [20,23] builds upon Hoare logic [13] and in the context of concurrent programs also on the Owicki-Gries method [21].

We assume a Java-like OO programming language that supports aliasing and references: allocation and deallocation of heap addresses (memory cells), as well as assignments to and from a heap memory cell. The main motivation behind the separation logic is to describe in a succinct way the state of the heap during program execution.

A separation logic assertion is interpreted on a program state (s, h), where s and h are a store and a heap, respectively. The store is a function mapping program variables to values and the heap is a partial map from pairs of object IDs and object fields to values. A value is either an object or a constant. To capture the heap related aspects, separation logic extends the syntax and semantics of the assertional part of Hoare logic. Separation logic adds heap operators (expressions) to the usual first order assertions of Hoare logic. The basic heap expressions are **emp**, i.e., the empty heap, satisfied by states having a heap with no entries, and $E \mapsto F$ (read as "E points to F"), i.e., a singleton heap, satisfied by a state with a heap consisting of only one entry at address E with content F. For instance, $o.x \mapsto v$ means that field x of object o has value v. Two heap expressions H_1 and H_2 corresponding to heaps h_1 and h_2, respectively, can be combined using the *separating conjunction operator* $*$, provided h_1 and h_2 have disjoint address domains. Expression $H_1 * H_2$ corresponds to the (disjoint) union $h_1 \uplus h_2$ of the heaps. Note that H_1 and H_2 describe two separate parts of the heap, h_1 and h_2, respectively. In contrast, the standard conjunction $p_1 \wedge p_2$, where p_1 and p_2 are separation logic formulae, corresponds to the whole heap satisfying both p_1 and p_2. Because of the domain disjointness requirement, the separation logic formula $(o.f \mapsto 10) * (o.f \mapsto 10)$ evaluates to **false**, whereas $(o.f \mapsto 10 \wedge o.f \mapsto 10)$ is equivalent to $(o.f \mapsto 10)$.

Like in Hoare logic, the triple $\{P\}C\{Q\}$, where C is a (segment of) a program and P and Q are assertions describing its pre- and post-condition, respectively, only concerns partial correctness; termination of C needs to be proven separately.

Separation logic adds to the standard rules (axioms) of the Hoare framework axioms for each of the new statements - allocation, deallocation, and assignments involving the heap cells. An important characteristic of separation logic is *tight interpretation*.

In some cases it is needed to embed a precise specification of a program segment C into a more general context. A specific axiom that allows this by enlarging the specification of a program segment C with an assertion R describing a disjoint heap segment which is not modified by any statement in C, is the *frame rule*:[1]

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}$$

In a concurrent setting, the programming language is extended with a `fork` statement, allowing to run program components in separate threads. For synchronized access to global objects, semaphores are added, together with the corresponding methods `acquire` and `release`.

The Owicki-Gries method extends the Hoare approach to concurrent programs preserving modularity. The first idea is to capture component behavior with non-shared ghost variables enabling separate proofs of concurrent components (for more on ghost variables, see Sect. 4). The second idea is to link shared resource and ghost variable values through an invariant that holds outside critical regions [21]. A resource A is a set of heap locations, and I_A is its associated invariant. The crucial idea is that each component may update or access these locations only within critical regions [11,14] in which the component has exclusive access to the locations. Although I_A may be violated within the critical region, it is guaranteed to hold at the beginning and at the end of the critical region. This is reflected in the rules for `acquire` and `release`:

$$\{\mathbf{emp}\}\texttt{s.acquire()}\{I_A(s)\}$$
$$\{I_A(s)\}\texttt{s.release()}\{\mathbf{emp}\}$$

The above described approach allows compositional verification. Each method m belonging to a class C is verified as a sequential program considering the invariants as extra constraints. Class C is considered verified when all its methods are verified. Since the program can be seen as a combination of classes and declarations, the whole program is verified when all its classes are verified.

One of the central concepts in concurrent separation logic is *ownership*. Due to tight interpretation, separation logic assertions can describe precisely the heap "footprint" of a given program C, i.e., the parts of the heap which C is allowed

[1] Here we disregard the usual side condition of the frame rule, since we assume a Java-like programming language not supporting global variables.

to use. Let l be a program component location and E a heap address. The component owns address E at location l iff E is contained in a heap corresponding to an assertion H which is true at location l. If $E \mapsto F$ is part of the heap corresponding to H, then this can be seen as an informal *permission* [5] for the verified component to read, write or dispose of the contents of the heap cell at address E.

Partial permissions are introduced to allow shared ownership of variables. Ownership is split into a number of fractional permissions, each of which only allows read only access. Expression $E \mapsto F$ denotes permission 1, i.e., exclusive ownership, whereas a fractional permission is expressed as $[z]E \mapsto F$ with $0 < z < 1$. Expression $[1]E \mapsto F$ is equivalent to $E \mapsto F$. Permissions can be split and merged during a proof. For instance, two fractional permissions can be merged according to the following rule: $[z]E \mapsto F * [z']E \mapsto F$, where $z + z' \leq 1$, implies $[z + z']E \mapsto F$. One acquires full ownership (and therefore write access) in case $z + z' = 1$. The split rule is analogous.

We use fractional permissions to enforce the syntactic rules and side conditions of Owicki-Gries on the use of the (global) real and ghost variables. For instance, by acquiring a semaphore, a component acquires the semaphore invariant. The semaphore invariant provides full permission to change the real variables and the ghost variables associated with the component. A component always holds a fraction of the permission for its ghost variables, thereby ruling out that other components change them. When releasing the semaphore, the component also releases the acquired (partial) ownerships.

4 Modular Specification Schema

Our aim is to specify modelling constructs and verify the implementation of those constructs in a modular way, meaning that each construct and its implementation should be independently specifiable and verifiable. The benefits of a modular approach are (1) that it will scale better than a monolithic approach and (2) that once a construct has been specified, we can abstract away its implementation details when verifying properties of the system.

It is crucial that implementations of constructs do not need to be verified again when their context changes. Because of this, and the fine-grained nature of the generic code, standard methods like Owicki-Gries do not suffice. In [16], a modular specification schema was proposed to solve this problem. In this section, we introduce an improved version of this modular approach which, compared to [16], provides a better abstraction from the implementation of the verified method.

As already mentioned, the Java methods in our transformation framework implement fine-grained parallelism. This means that each method may acquire and release access to multiple critical regions (CR) during its execution, instead of following a coarse-grained approach in which the complete method is executed in one big CR. As CRs tend to form performance bottlenecks in software, a fine-grained approach tends to decrease the level of dependency between threads in a multi-threaded system, and thereby increase the overall performance.

In order to verify methods with fine-grained parallelism, so-called *ghost code* must be inserted as part of the annotations. To see how this mechanism of code insertion works, we consider a method m belonging to a class `C` instantiated in an object `o`. We want to give a specification of m in the form of a standard Hoare logic triple $\{P\}\texttt{o}.m\{Q\}$. Under fine-grained parallelism one cannot formulate P and Q in terms of the actual fields determining the state of `o`. For instance, consider method `send(msg, G)` that sends a message `msg` to a channel queue `q` (`q` is a field of `C`), as in Listing 1. At line 8, the piece of code `G` given as a parameter to `send` is inserted.

Listing 1. A fine-grained `send` operation

```
1  class C
2     queue q
3     semaphore s
4     method send(msg,G)
5     begin
6        s.acquire()
7        q := q + msg
8        G
9        s.release()
10    end
```

In a concurrent setting, multiple threads may send messages to the queue of a single instance of `C`. In that case, `q` may be changed by some other `send` call between the call to `send(b)` leaving the CR protecting `q` (line 9) and `send(b)` returning the control to the calling client program (line 10). We cannot claim that once `send(a)` is finished, the new content of `q` is `q+a`, where `+` indicates concatenation. This is analogous to Owicki-Gries, where global variables altered by multiple modules cannot be used directly to specify a module.[2]

To resolve this, ghost variables (also called logical or auxiliary variables) are added to the program. Ghost variables are write-only, i.e., the instrumented program can change them, but not read them. Hence, they do not change the control flow of the program and are only auxiliary verification devices. Each ghost variable is owned by a particular process, and only this process can potentially change its content. To illustrate the use of ghost variables, let us assume that `send` is used by a client program as shown in Listing 2.

Listing 2. A client using the `send` method.

```
1  o := New C()
2  o.send(a) || o.send(b)
```

Suppose we want to prove that if in the beginning of the program $len(\texttt{q}) = 0$ holds, where *len* gives the length of the queue, then at the end, $len(\texttt{q}) = 2$. We specify the two instances of `send` by introducing ghost variables y and z to capture the local effect on the length of `q` in the left and right method call, respectively. Resource invariant $I_A \equiv len(\texttt{q}) = y + z$ captures how these local effects relate to the global resource. Now we can specify `send(a)` with $\{y = 0\}\texttt{send(a)}\{y = 1\}$ and `send(b)` with $\{z = 0\}\texttt{send(b)}\{z = 1\}$. Finally, we define

[2] In the classical Owicki-Gries framework this is directly forbidden by the interplay of the syntactic rules of the usage of the global variable and the side conditions of the axioms for CR and parallel composition.

$G \equiv y := 1$ for `send(a)` and $G \equiv z := 1$ for `send(`b`)`, to update y and z, respectively, at line 8 in Listing 1 when `send` is executed.

With verification axioms similar to Owicki-Gries, it can be proved that these assertions indeed confirm the correctness of the client property. In particular, the conjunction of the postconditions of `send(a)` and `send(b)`, and I_A, i.e., $y = 1 \wedge z = 1 \wedge len(\mathtt{q}) = y + z$, implies the desired client postcondition $len(\mathtt{q}) = 2$.

Passing corresponding ghost codes G to instances of m allows for abstraction and parallelism, but it does not make the approach modular. Each context and/or property likely requires different ghost variables, and hence different P, Q, I_A, and G. Suppose that we want to verify a property about the content of `q` using a function *cnt* mapping the queue content to a set of messages. Specifically, we want to prove that if in the beginning, $cnt(\mathtt{q}) = \emptyset$, then at the end, $cnt(\mathtt{q}) = \{\mathtt{a}, \mathtt{b}\}$. In this case, our ghost variables range over sets of messages, and the specifications must be adjusted accordingly, i.e., $\{y = \emptyset\}\mathtt{send(a)}\{y = \{\mathtt{a}\}\}$, $\{z = \emptyset\}\mathtt{send(b)}\{z = \{\mathtt{b}\}\}$, $I_A \equiv cnt(\mathtt{q}) = y \cup z$, and $G \equiv y := \{\mathtt{a}\}$ and $G \equiv z := \{\mathtt{b}\}$ for `send(a)` and `send(b)`, respectively. Even if we had a library of predicate sets and ghost code blocks, in general we would not be able to cover all possible contexts in which the generic code, i.e., m, could be used.

Greater generality can be achieved by a schema along the lines of [16] in which P, Q, I_A, and G are parameters of the specification of m. The schema imposes some constraints on these parameters which become proof obligations when verifying code involving m. Under these constraints, m needs to be verified only once. For each new context, the client only needs to verify that the contraints hold. We propose a new modular specification schema (MSS) that allows further abstraction from the implementation details of m, by supporting parameterization based on CRs. Unlike in [16], the semaphores that implement the CR as well as the names of the fields that determine the state of the object (s and q, resp., in the `send` example) remain absent from the specification. As a result one retains the flexibility of the OO approach. For example, if the implementation of the CR is changed such that locks are used instead of semaphores, the specification can remain the same.

We proceed by giving the intuition behind the MSS. We first establish the relationships between the parameters P, Q, I_A, and G, that need to hold in order for the specification to be correct. Later we lift these relationships to the level of the whole method m to formulate the MSS.

Listing 3. A semaphore based implementation of a CR

```
1  {P}
2  s.acquire()
3  {IA(s) * P}
4  {O(v) * I(v) * P}
5  C
6  {O(post(v)) * I(v) * P}
7  G
8  {O(post(v)) * I(post(v)) * Q}
9  {IA(s) * Q}
10 s.release()
11 {Q}
```

Assume that the body of m consists of only a single CR implemented by using semaphore `s`. The CR is of the form `s.acquire()` C `s.release()` as given in Listing 3. Without loss of generality, let us assume that the CR protects a single field `f` of an instance `o` of class `C`. Field `f` can be changed only within the CR.

When establishing the relationships, we are guided by the correctness requirements for the annotation of Listing 3 in the familiar Hoare logic/Owicki-Gries style. The validity of P and Q at lines 1 and 11, respectively, implies that $I_A(\mathtt{s}) * P$ and $I_A(\mathtt{s}) * Q$ hold at lines 3 and 9, respectively (we write $I_A(\mathtt{s})$ instead of just I_A to emphasize that it is associated with `s`). This follows from the rules from Sect. 3 (for `acquire` and `release` combined with the frame rule), and the fact that P and Q do not refer to `s` and hence involve parts of the heap disjoint from the parts affected by `acquire` and `release`. This is analogous to the proof rule for the CR in Owicki-Gries.

To capture the environment constraints, next to ghost variables, $I_A(\mathtt{s})$ may also depend on `o.f`. To avoid directly referring to `f` and thereby making the approach modular, we introduce a so-called *payload invariant* I, parameterized with a ghost variable v. In the example of Listing 2, $I_A(\mathtt{s}) \equiv len(\mathtt{q}) = y + z$ would be substituted by $I(v) \equiv len(v) = y + z$. To link the actual field `f` with its ghost counterpart v we use predicate $O(v)$ (for the earlier `send` example, we could define $O(v) \equiv \mathtt{q} = v$). $O(v)$ is an abstract predicate local to `o` that is not visible for the client. By defining $I_A(\mathtt{s}) = \exists v.O(v) * I(v)$, we circumvent the need to refer to `o.f` in the client invariant.

Line 4 in Listing 3 is obtained by substituting $O(v) * I(v)$ for $I_A(s)$ at line 3. Since C affects only actual variables, P holds also in the postcondition of C at line 6. However, since the actual variables have changed while ghost variable v remains the same, predicate O holds only for an adjusted value of v given by $post(v)$. In our example, $post(v) \equiv len(v) + 1$. G only affects y and z, so after G, $O(post(v))$ remains valid. So, in order to recover the invariant I_A, G at line 7 should be chosen such that it modifies the ghost variables occurring in $I(v)$ and P in such a way that $I(post(v))$ becomes **true** and P is transformed to Q (line 8). Proving that G indeed has this property remains a proof obligation for the client program calling m and as such becomes a premise of our schema. It is easy to check that this constraint is satisfied by all instances of `send` in the running example for both client properties. Finally, line 9 follows directly from line 8 by the definition of $I_A(\mathtt{s})$.

The Modular Specification Schema. By summarizing the constraints on the various elements of the annotation, and lifting them to the level of method m, we obtain the MSS:

$$\frac{\forall v \bullet \{P * I(v)\}\ G\ \{Q(res(v)) * I(post(v))\}}{\{\exists v \bullet O(v) * [\pi]\mathtt{o}.A(I(v)) * P\}\ \mathtt{r} := \mathtt{o}.m(G)\ \{\exists v \bullet O(v) * [\pi]\mathtt{o}.A(I(v)) * Q(\mathtt{r})\}}$$

For simplicity, we assume that m has no parameters besides G. However, additional parameters can be captured in the usual way for procedure verification rules in Hoare logic. We also assume that m returns a result $res(v)$ immediately

after leaving the CR, that is assigned to variable `r`. In general, Q depends on `r`. Both $res(v)$ and $post(v)$ are fixed by the supplier of m.

Predicate A links semaphore `s` with the payload invariant I. Both A and O are abstract predicates in the sense that the client does not need to know their definition since they are local to `o`. For the `send` example, A would state that there is a semaphore `s` that is properly initialized and it associates to A a semaphore invariant $I_A(\texttt{s})$ (formed using $I(v)$ as described earlier). These implementation details, including `s`, are hence not visible to the client calling m. Finally, π is an arbitrary fraction denoting a fractional permission for A.

Note that MSS is not an axiom or a proof rule of separation logic, since for any correct module it can be derived from other axioms and rules. The correctness of MSS can be verified using the annotation in Listing 3.

MSS can be seen as a means to divide the proof obligations between the client and the supplier of m. The schema is implicitly universally quantified over P, Q, I, and G. Note that *post* and *res* are fixed by the supplier and that they implicitly define the effect of C on `o.f` in a sequential environment. On the other hand, the client is free to use any predicates P, Q, I, and G satisfying the premise of MSS. For any such predicates, the supplier guarantees that the implementation of m satisfies the triple in the consequent of MSS.

The premise of MSS $\forall v.\{P * I(v)\}\ G\ \{Q(res(v)) * I(post(v))\}$ is analogous to the premise of the Owicki-Gries CR axiom $\{P * I_A(\texttt{s})\}\ C\ \{Q(\texttt{r}) * I_A(\texttt{s})\}$. MSS, however, shifts the verification from the actual code C and invariant I_A to the ghost code G and the payload invariant I. Although C does not appear in MSS, its specification is reflected in v, $post(v)$ and $res(v)$. Although G has to reflect all important aspects of each call of `o.m`, the method is still to a great extent modular since the implementation and verification of the program text of $o.m$ remains completely independent of the call of `o.m` which is invoked.

The soundness of the modular schema follows from the same arguments presented in [16].

5 Specifying and Verifying the SLCO Channel

In this section we present the specification and verification of an essential part of the generic code for our SLCO-to-Java transformation, namely the communication channel. We specify the channel for use in a generic, multi-threaded environment. Using VeriFast, we verify the absence of race conditions and deadlocks, and show how to prove properties of clients using the channel.

SLCO models use asynchronous non-blocking lossless channels that can hold a predefined maximum number of messages. The channel datastructure provides two operations, `send` and `receive`, to add and remove messages. It has a FIFO structure, i.e., messages are added to the end and removed from the front of a queue. Provided that the client program invoking a channel operation has exclusive access to the channel, the specification of the operations is as follows. The `send` operation has one parameter `msg`, the message that is being sent. If the contents of the channel is `q` and it is not full when `send` is started, then

after execution of `send` the contents of the channel is `q + msg`, where `+` denotes concatenation of sequences of messages. Furthermore, `send` returns a Boolean result indicating whether or not the operation was successful; if the channel is already full when `send` is started, **false** is returned. Whenever `receive` is started and the channel has contents `msg + q`, then the channel's new contents after execution of `receive`, provided that any provided conditions hold, is `q`, and message `msg` is returned as a result. If the channel is empty when `receive` starts executing, then `receive` is blocked until it succeeds to remove a message. Since the channel is used in a multi-threaded environment, adding and removing messages should be done atomically.

We illustrate our modular approach described in Sect. 4 on the `send` method of the channel implementation. In VeriFast, each Java source code file being verified is linked to a specification file only containing (abstract) predicates and specifications of Java methods and ghost functions. The VeriFast specification of the method following MSS is given in Listing 4. (The complete specification and annotated implementation files will become part of the Java examples set in the standard distribution of VeriFast.)

Listing 4. Part of the channel specification

```
1  public final class Channel {
2      //...
3
4      boolean send(String msg)
5      /*@
6      requires
7          [?pi]A(?I) &*&
8          is_G_S(?G, this, I, msg, ?P, ?Q) &*& P();
9      @*/
10     /*@
11     ensures
12     [pi]A(I) &*& Q(result);
13     @*/
14    // ...
15 }
```

The VeriFast specific text, i.e., specifications and ghost variable declarations, is inside special comments delimited by `@`. The pre- and postconditions that form the contract are denoted by the keywords `requires` and `ensures`, respectively. Component predicates of the pre- and postcondition are glued by the separating conjunction operator denoted by `&*&`. Predicates `A`, `I`, `P`, and `Q` correspond to their namesakes in the MSS, whereas the assertion `is_G_S` implements the passing of the ghost code G into the method. Both `[?pi]` and `[pi]` correspond to the fractional permission $[\pi]$. The question mark `?` in front of a variable means that the value of the variable is recorded and that all later occurrences of that variable in the contract must be equal to the first occurrence. For instance, in Listing 4, the value of the fractional permission `pi` in the precondition must be the same as the one in the postcondition (as also required in the MSS).

Predicates `P`, `Q` and `is_G_S` are left undefined and are supposed to be provided by the client. More precisely, a *lemma function* `G` is supplied by the client based on which VeriFast automatically creates the predicate `is_G_S`. A VeriFast lemma function is a method without side effects which helps the verification engine.

The contract of a lemma function corresponds to a theorem, its body to the proof, and a lemma function call to an application of the theorem. Listing 5 contains the specification of `G` that corresponds to the ghost statement block G.

Note that the specification of `G` in Listing 5 corresponds to the premise of MSS, where $post(v)$ specifies that if res = **true**, `msg` has been added to the channel, and otherwise it has not (line 4).

Listing 5. A lemma function specifying the ghost statement block G

```
1 /*@
2 typedef lemma void G(Channel c, predicate(list<Object>, int) I,
      String msg, predicate() P, predicate(boolean) Q)(boolean res);
3     requires P() &*& I(?items, ?qms);
4     ensures Q(res) &*& I(res ? append(items, cons(msg, nil)) :
          items, qms);
5 @*/
```

Method `send` is part of the class `Channel` (Listing 6), implementing the SLCO channel construct. Class `Channel` contains three fields: the list `itemList` implementing the FIFO queue, semaphore `s` that is used to implement access to the CR within the operations, and `queueMaxSize` defining the maximum channel capacity. For verification purposes we add the ghost field `inv` which is used to keep track of the invariant.

Semaphore invariant `I_A`, corresponding to I_A in Sect. 4, is given at lines 3–4 in Listing 6. The invariant is defined by means of a predicate constructor parameterized with the payload invariant `I`. Corresponding to the definition of I_A, in `I_A`, it is checked that for ghost variables `items` and `qms`, i.e., the contents of the item list and the maximum number of messages, respectively, `I` holds. The question mark `?` is used to record the value of the variable following it, for use later on in the predicate. Operator $\mapsto$ is written in VeriFast as `|->`, and the expression of the form `[f]` denotes fractional ownership with fraction `f`. When `f` = 1, the fractions are omitted, and an arbitrary fraction is denoted as `[_]`.

Listing 6. A specification of the Channel class

```
1
2 /*@
3 predicate_ctor I_A(Channel channel, predicate(list<Object>, int) I)
      () =
4     channel.O(?items, ?qms) &*& I(items, qms);
5 @*/
6
7 public final class Channel {
8     List itemList;
9     Semaphore s;
10    int queueMaxSize;
11    //@ inv inv;
12    //@ predicate O(list<Object> items, int qms) = this.itemList |->
          ?itemList &*& itemList.List(items) &*& this.queueMaxSize
          |-> qms &*& length(items) <= qms;
13    //@ predicate A(predicate(list<Object>, int) I) = ... &*& s |->
          ?sem &*& [_]sem.Semaphore(I_A(this, I));
14 }
```

In predicate `O` (line 12), as explained earlier, the links are established between ghost variables and fields. Its first conjunct `channel.itemList |-> ?itemList`

implies exclusive ownership of the field `itemList` and at the same time that the value of `itemList` is recorded for later use in the contract. Expression `itemList.List(items)` states the fact that `itemList` is a list with elements `items`. The final conjunct links `queueMaxSize` to ghost variable `qms`.

We use the VeriFast ownership concept to implement syntactic restrictions on the variables. In particular, we need to ensure that the fields like `itemList` can be modified only in the CR implemented by semaphore `s` and that the ghost variables are modified exclusively by at most one method, in this case `send`.

Predicate `A` is given at line 13 in Listing 6. Like its MSS counterpart A, it is parameterized with the payload invariant `I` (corresponding to I in MSS). Besides some auxiliary conjuncts, it has two conjuncts to associate `I_A` with `s`, the first of which is parameterized with the payload invariant and the object itself.

Listing 7. The annotation of the Channel `send` method

```
1  public final class Channel {
2      //...
3      public boolean send(String msg)
4      /*@ requires ... ensures ... @*/
5      {
6          //@ open [pi]A(I);
7          //@ s.makeHandle();
8          s.acquire();
9          //@ open I_A(this, I)();
10
11         boolean result = itemList.size() < queueMaxSize;
12         if (result)
13             itemList.add(msg);
14
15         //@ G(result);
16         //@ length_append(items, cons(msg, nil));
17         //@ close I_A(this, I)();
18         s.release();
19         //@ close [pi]A(I);
20         return result;
21     }
22     //...
23 }
```

Listing 8. Client program specification

```
1  public class Program {
2      //@ static int sendCount;
3      //@ static int receiveCount;
4      public static int messageMaxCount; // k
5
6      public static void main(String[] args)
7      //@ requires class_init_token(Program.class) &*&
           Program_messageMaxCount(?mmc) &*& 0 < mmc;
8      //@ ensures Program_messageMaxCount(mmc) &*& [_]
           Program_sendCount(?sc) &*& [_]Program_receiveCount(?rc) &*&
           mmc == sc &*& mmc == rc;
9      {
10         // ...
11     }
12 }
```

Listing 7 contains the `send` method with its corresponding full annotation that further facilitates verification. Since VeriFast does not automatically unfold predicate definitions, ghost statement `open` is used to do this, i.e., to replace

the predicate with its definition. In this way the heap chunks of the definition are made visible to the verifier. The opposite effect is achieved by `close` which replaces heap chunks with the corresponding predicate definition. At line 6 predicate `A` is unfolded to obtain the predicates needed for acquiring `s`. After the acquisition of the semaphore also its invariant `I_A` is opened at line 9 to get access to the heap chunks related to `itemList` and `queueMaxSize`.

The code segment at lines 11–13 corresponds to C in the MSS, and affects the "real" variables. The code at lines 15–17 is ghost code. The lemma function performing the updates of the ghost variables is called at line 15. Annotation of the `receive` method can be done in an analogous way.

Class `Channel` annotated as in Listing 7 is verifiable against its specification in VeriFast. This means that it is free of deadlocks and race conditions. Those requirements are not explicitly specified, but are always checked when VeriFast tries to verify code. The class is now ready to be used by client programs to verify specific properties, using the pre- and postconditions and the payload invariant.

Listing 9. SenderThread class specification

```
1  class SenderThread implements Runnable {
2      //@ predicate pre() = this.c |-> ?c &*& [_]c.A(I) &*& [_]
           Program_sendCount(0) &*& [_]Program_messageMaxCount(?mmc)
           &*& 0 < mmc;
3      //@ predicate post() = this.c |-> ?c &*& [_]c.A(I) &*& [_]
           Program_messageMaxCount(?mmc) &*& [_]Program_sendCount(?sc)
           &*& mmc == sc;
4
5      Channel c;
6      ...
7
8      public void run()
9      //@ requires pre();
10     //@ ensures post();
11     {
12         for (i = 0; i < Program.messageMaxCount; i++)
13         {
14           for (;;){
15           /*@
16                predicate P() = [1/2]Program_sendCount(i) &*& [1/3]
                       Program_messageMaxCount(mmc);
17               predicate Q(boolean r) = [1/2]Program_sendCount(r ? i
                       + 1 : i) &*& [1/3]Program_messageMaxCount(mmc);
18               lemma void ghost_send(boolean r)
19                 requires ... ensures ...
20               {
21                 open P();
22                 ...
23               }
24           @*/
25           //@ produce_lemma_function_pointer_chunk(ghost_send) : G_S
                   (c, I, m, P, Q)(r) { call(); };
26           //@ close P();
27           boolean success = this.c.send("message");
28           //@ open Q(success);
29           }
30         }
31         //@ close post();
32     }
33 }
```

Next, we discuss how the property 'if k messages are sent over the channel, k messages will be received' can be specified for a program using the channel via one sending and one receiving thread. First, of all, Listing 8 specifies the client program we use. In the `main` method (lines 6 and onwards), an instance of the channel is created, and a sending and a receiving thread are started, one sending k, i.e. `messageMaxCount`, messages, and the other one trying to receive them. To specify the property, we introduce two new ghost variables for counting the number of messages (lines 2 and 3). In the precondition of `main`, we require that the class has been properly initialized (conjunct 1 at line 7), link the `messageMaxCount` variable to the ghost variable `mmc`, and have an additional requirement that it is at least equal to 1. In the post-condition, we link `sendCount` and `receiveCount` respectively to `sc` and `rc`, and require that they are both equal to `mmc` (line 8).

To determine that the post-condition holds, we need to specify the thread sending the messages. In Listing 9, at lines 2 and 3, its pre- and post-condition are specified. In the `run` method, the messages are sent. For the `send` call at line 27, we need to provide ghost code `G_S`. This is done in lemma `ghost_send`, where the ghost variables are updated. This lemma is linked to the call at line 25. The pre- and post-condition of `send` are specified as two predicates, `P` and `Q`, see lines 16 and 17.

VeriFast was able to verify the code against its specification, meaning that the property holds. Besides the environment with two threads, we were also able to consider an environment consisting of multiple senders and receivers, to verify that no conflicts can arise in such a setting.

6 Related Work

Much work has been done and continues to be done on the verification of model transformations. For an overview of the field, see [22]. Here, we mention some relevant work that also focusses on (1) model-to-code transformations, (2) formal verification of correctness using theorem provers, and (3) correctness as the preservation of behavioural semantics. The latter seems to be the most relevant interpretation of correctness mentioned in [22], as it addresses behavioural aspects, in our case, for example, race and deadlock freedom of communication channels, both in SLCO and the Java implementation.

Amphion [2] is a tool to generate code from models of space geometry problems. It uses a theorem prover automatically, hiding the details from the user, to create Fortran source code that is correct by construction. Besides addressing a different type of models, they do not separately consider the generic code constructs used. We, on the other hand, have yet to prove correctness of our entire transformation method. It would be interesting to see if their approach is to some degree applicable for us.

In [24], the QVT language and transformations are formalised for use with the KIV theorem prover, to verify Java code generators for security properties and syntactic correctness. Their approach is operational, but scalability is still a

serious issue. We wonder whether a split similar to ours of the proof obligations for generic and specific code would improve the scalability.

Other techniques address very similar issues, but work strictly indirectly, i.e., they focus on code generated from a concrete model as opposed to transformations that produce code. We mention some works here, since our work can to some extent also be considered as indirect (one condition for directness given in [22] is that the transformation rules are formalised, which we have not done yet). Blech [4] verifies semantics preservation of a statechart-to-Java transformation using Isabelle/HOL. In [9,10], annotations are generated together with code to assist automatic theorem proving. The latter is a very interesting approach that we may consider for the analysis of our specific code.

An approach to generate Java code from Communicating Sequential Processes (CSP) specifications is described in [27]. The authors describe how they have verified that a CSP model of their implementation of a channel semantically corresponds with a simpler CSP model describing the desired functionality of that channel. First of all, by working from a model describing the implementation, as opposed to the implementation itself, one still needs to prove that the model corresponds exactly with the implementation to establish that the implementation itself is correct. Second of all, it seems that a fully modular verification approach in the way we wish to have it is not completely possible; for instance, although it would be possible to use their simpler CSP model of a channel within detailed implementation-level CSP models of systems using channels, one could not abstract away the functionality of a channel to the same extent as when using separation logic if one would like to prove a functional property referring to communication, but not expressing how the communication itself should proceed.

Regarding theorem proving, to the best of our knowledge the approach in [16] was the first one supporting fully general modular specification and verification of fine-grained concurrent modules and their clients. Compared to the schema in [16], the MSS we propose imposes conditions on the ghost code instead of the actual code, and abstracts away the implementation of the protected object better than [16] does, thereby improving the modular nature of the approach.

An approach comparable to [16] appears in [26] where a new separation logic is presented with concurrent abstract predicates. Furthermore, in [25] they have applied their approach to prove correctness of some synchronisation primitives of the Joins concurrent C# library. As far as we know, the authors do not intend to eventually use their approach to verify model transformations. It remains to be investigated whether theirs can be used for that as well.

Another viable option to verify model-to-code transformations seems to be the use of software model checking techniques, in which a formalization of a program is checked against an automaton capturing a specification [7,17]. However, it remains to be investigated whether one can verify implementations of modelling constructs for general environments as we have done here.

The Java Modelling Language (JML) is a behavioural interface specification language for Java. An advantage of JML over separation logic is that Java expressions can be used. Several verification tools have been developed that

use JML as a specification language [6]. The extended static checker for Java (ESC/Java2) [8], for instance, was one of the first of such tools. However, it is not designed to prove full functional correctness, but rather find common programming errors, and hence it is not suitable for our task. Krakatoa [19] and the Key tool [3], on the other hand, are program verifiers that may be used by us as alternatives to VeriFast. To which extent this is possible remains to be investigated.

Adding ownership types [12,33] to Java is a very effective technique to verify that Java threads always access data correctly, i.e. for which they have acquired the proper access rights. Such a technique offers an alternative way to verify that our channel implementation is always correctly accessed. However, it cannot be used to verify arbitrary functional properties that may rely on ownership, but express more than that, such as that some desired behaviour is guaranteed to always eventually happen. On the other hand, with separation logic, one can express and verify such properties as well.

7 Conclusions

We introduced an MDSE approach where generated code is separated into a generic and a model specific part. We presented an application of a modular approach for the verification of fine grained concurrent code in this context using the VeriFast tool. This paper showed the ideas behind and the feasibility of such an approach. With its support of parameterized verification, concurrency via threads, object-oriented code, and fast verification results,VeriFast was up to the task - though an experienced user is required. This underlines the relevance of the idea of re-using generic code that has to be verified only once.

We introduced a novel module specification schema which improves the modularity of the VeriFast approach. Although the schema was originally developed having in mind separation logic and VeriFast, it can be straightforwardly adapted for the standard Owicki-Gries method (assuming extensions with modules) or similar formalisms for concurrent verification.

Finally, using theorem provers to verify the correctness of code still requires considerable expert knowledge. We observe that by using model-to-code transformations, experts can focus on proving correctness of those transformations, thereby relieving developers from the burden to prove that code derived from specific models is correct.

In future work, we plan to address liveness issues, both in the framework and as regards verification, and we plan to address verification of the complete model-to-code transformation, i.e., not only that the used generic code constructs are correct, but that it is guaranteed that the complete executable code is always correct. This is quite challenging, since SLCO also supports the timing of actions. SLCO models with timing can be formally verified by first discretising the timing [31]. Other relevant challenges and ideas are reported in [32].

Acknowledgments. We would like to thank Suzana Andova for the discussions in the early phases of the work described in this paper.

References

1. van Amstel, M., van den Brand, M., Engelen, L.: An exercise in iterative domain-specific language design. In: EVOL/IWPSE. pp. 48–57. ACM (2010)
2. Baalen, J.V., Robinson, P., Lowry, M., Pressburger, T.: Explaining synthesized software. In: ASE. pp. 240–248. IEEE (1998)
3. Beckert, B., Hähnle, R., Schmitt, P. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
4. Blech, J., Glesner, S., Leitner, J.: Formal verification of java code generation from UML models. In: Fujaba Days, pp. 49–56 (2005)
5. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. ACM SIGPLAN Not. **40**(1), 259–270 (2005)
6. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G., Leino, K., Poll, E.: An overview of JML tools and applications. STTT **7**(3), 212–232 (2005)
7. Chaki, S., Clarke, E., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. In: ICSE, pp. 385–395. IEEE (2003)
8. Leavens, G.T., Poll, E., Kiniry, J.R., Chalin, P.: Beyond assertions: advanced specification and verification with JML and ESC/Java2. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 342–363. Springer, Heidelberg (2006)
9. Denney, E., Fischer, B.: Generating customized verifiers for automatically generated code. In: GPCE, pp. 77–88. ACM (2008)
10. Denney, E., Fischer, B., Schumann, J., Richardson, J.: Automatic certification of kalman filters for reliable code generation. In: IEEE Aerospace Conference. pp. 1–10. IEEE (2005)
11. Dijkstra, E.W.: Cooperating sequential processes. In: Brinch Hansen, P. (ed.) The Origin of Concurrent Programming. From Semaphores to Remote Procedure Calls, pp. 65–138. Springer, New York (2002)
12. Fogelberg, C., Potanin, A., Noble, J.: Ownership meets java. In: IWACO, pp. 30–33 (2007)
13. Hoare, C.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
14. Hoare, C.A.R.: Towards a Theory of Parallel Programming. In: Brinch Hansen, P. (ed.) The Origin of Concurrent Programming. From Semaphores to Remote Procedure Calls, pp. 231–244. Springer, New York (2002)
15. Jacobs, B.: VeriFast website. people.cs.kuleuven.be/∼bart.jacobs/verifast/ (2012)
16. Jacobs, B., Piessens, F.: Expressive modular fine-grained concurrency specification. In: POPL, pp. 271–282. ACM (2011)
17. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. **41**(4), 21:1–21:54 (2009)
18. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture(TM): Practice and Promise. Addison-Wesley Professional, Boston (2005)
19. Marché, C., Paulin-Mohring, C., Urbain, X.: The krakatoa tool for certification of java/javacard programs annotated in JML. J. Logic Algebraic Program. **58**(1–2), 89–106 (2004)
20. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)
21. Owicki, S., Gries, D.: Verifying properties of parallel programs: an axiomatic approach. Commun. ACM **19**(5), 279–285 (1976)

22. Rahim, L., Whittle, J.: A survey of approaches for verifying model transformations. Softw. Syst. Model. **14**(2), 1003–1028 (2015)
23. Reynolds, J.: Separation logic: a logic for shared mutable data structures. In: LICS, pp. 55–74. IEEE (2002)
24. Stenzel, K., Reif, W., Moebius, N.: Formal verification of QVT transformations for code generation. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 533–547. Springer, Heidelberg (2011)
25. Svendsen, K., Birkedal, L., Parkinson, M.: Joins: a case study in modular specification of a concurrent reentrant higher-order library. In: Castagna, G. (ed.) ECOOP 2013. LNCS, vol. 7920, pp. 327–351. Springer, Heidelberg (2013)
26. Parkinson, M., Birkedal, L., Svendsen, K.: Modular reasoning about separation of concurrent data structures. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 169–188. Springer, Heidelberg (2013)
27. Welch, P., Martin, J.: Formal analysis of concurrent java systems. In: CPA, pp. 275–301. IOS Press (2000)
28. Wijs, A.: Define, verify, refine: correct composition and transformation of concurrent system semantics. In: Fiadeiro, J.L., Liu, Z., Xue, J. (eds.) FACS 2013. LNCS, vol. 8348, pp. 348–368. Springer, Heidelberg (2014)
29. Wijs, A., Engelen, L.: Efficient property preservation checking of model refinements. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 565–579. Springer, Heidelberg (2013)
30. Engelen, L., Wijs, A.: REFINER: towards formal verification of model transformations. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2014. LNCS, vol. 8430, pp. 258–263. Springer, Heidelberg (2014)
31. Wijs, A.: Achieving discrete relative timing with untimed process algebra. In: ICECCS, pp. 35–44. IEEE (2007)
32. Zhang, D., Bošnački, D., van den Brand, M., Engelen, L., Huizing, C., Kuiper, R., Wijs, A.: Towards verified java code generation from concurrent state machines. In: AMT, CEUR Workshop Proceedings, vol. 1277, pp. 64–69 (2014). CEUR-WS.org
33. Zibin, Y., Potanin, A., Li, P., Ali, M., Ernst, M.: Ownership and immutability in generic java. In: OOPSLA. ACM SIGPLAN Notices, vol. 45, pp. 598–617. ACM (2010)