Prototyping a Concurrency Model

Benjamin Morandi, Mischael Schill, Sebastian Nanz, Bertrand Meyer
Chair of Software Engineering, ETH Zurich, Switzerland
firstname.lastname@inf.ethz.ch
http://se.inf.ethz.ch/

***Abstract*—Many novel programming models for concurrency have been proposed in the wake of the multicore computing paradigm shift. They aim to raise the level of abstraction for expressing concurrency and synchronization in a program, and hence to help developers avoid programming errors. Because of this goal, the semantics of the models themselves becomes ever more complex, increasing the risk of design flaws. We propose a methodology for prototyping concurrency models using executable formal specifications. The prototype can be used to test and correct the model at an early stage. Once the development is completed, the executable formal specification serves as an unambiguous reference. We apply this methodology to SCOOP, an object-oriented concurrency model. Using this approach, we were able to uncover and fix three major flaws in the model.**

***Keywords*-formal specification; prototyping; testing; concurrent programming; SCOOP; Maude**

I. Introduction

A variety of concurrency models have been introduced to improve the development of concurrent programs. Examples include the Actor model [1], Communicating Sequential Processes (CSP) [2], the Join Calculus [3], the Partitioned Global Address Space (PGAS) model [4], the tuple space model [5], and SCOOP [6]. These models introduce new programming abstractions with sophisticated semantics, raising the possibility of design flaws.

Flaws in a concurrency model can have costly consequences if discovered after the model has been embedded into a programming language. It is therefore beneficial to verify the model prior to developing compiler and runtime support. Static verification, for example using theorem proving (e.g., [7], [8]) or model checking (e.g., [9], [10]), can be very time-consuming or otherwise only applicable to a small subset of the model.

This paper suggests instead to first develop an *executable* formal specification that serves as the prototype of the model. The model can then be verified dynamically: the designers use test programs to check whether the formal executions conform to expectation. This approach preserves on the one hand the rigor of a formal model but combines it with the simplicity of a testing approach. It cannot ultimately prove the absence of design flaws, and does not strive to do so; however, similarly to the role of testing in software development, we found it to be essential in debugging a model and as a prototyping tool.

The methodology involves the following sequence of steps:

1) Start with an informal description of the main characteristics of the model.
2) Develop an executable formal specification from the informal description. This rigorous step provides a first opportunity to discover and resolve flaws in the model. Several frameworks support the development of executable formal specifications. We use Maude [11], [12] because of its expressiveness and performance.
3) Use the executable formal specification to test the model: provide test programs and use the specification to analyze their executions; address the design flaws directly in the specification.

Once the development is completed, the executable formal specification serves as an unambiguous reference, which provides a starting point for future model extensions as well as for implementing compiler and runtime support.

This paper demonstrates the proposed methodology on SCOOP (Simple Concurrent Object-Oriented Programming) [6], [13], an object-oriented concurrency model. The resulting executable formal specification has more than 3100 lines of Maude code that cover the model in its details; it can be downloaded from [14]. The effort led to the discovery and elimination of three major design flaws. In follow-up work, the methodology again proved helpful in extending the SCOOP model with an asynchronous exception mechanism [15].

While the importance of verifying formal specifications of programming models is being recognized (see Section VI), to the best of our knowledge, this paper is the first to use a comprehensive executable formal specification to test and improve a concurrency model.

The remainder of the paper traces the steps outlined above. Section II describes SCOOP informally. Section III presents the formal specification, and Section IV shows its implementation in Maude. Section V shows how testing helped to resolve three flaws in SCOOP. Finally, Section VI discusses related work, and Section VII concludes with an outlook on future work.

II. Informal description of SCOOP

This section describes the SCOOP concurrency model informally. It also outlines one possible embedding into a programming language to support the presentation of the model. The full description is available in [6], [13], [16].

A. Introduction to SCOOP

The starting idea of SCOOP is that every object is associated for its lifetime with a processor, called its *handler*. A *processor* is an autonomous thread of control capable of executing

actions on objects. An object's class describes the possible actions as *features*.

A variable x belonging to a processor can point to an object with the same handler (*non-separate object*), or to an object on another processor (*separate object*). In the first case, a *feature call x.f* is *non-separate*: the handler of x executes the feature synchronously. In this context, x is called the *target* of the feature call. In the second case, the feature call is *separate*: the handler of x, i.e., the *supplier*, executes the call asynchronously on behalf of the requester, i.e., the *client*. The possibility of asynchronous calls is the main source of concurrent execution. The asynchronous nature of separate feature calls implies a distinction between a feature call and a *feature application*: the client logs the call with the supplier (feature call) and moves on; only at some later time will the supplier actually execute the body (feature application).

The producer-consumer problem serves as a simple illustration of these ideas. A root class defines the entities *producer*, *consumer*, and *buffer*.

```
producer: separate PRODUCER
consumer: separate CONSUMER
buffer: separate BUFFER [INTEGER]
```

The keyword **separate** specifies that the referenced objects may be handled by a processor different from the current one. A *creation instruction* on a separate entity such as *producer* will create an object on another processor; by default the instruction also creates that processor.

Both the producer and the consumer access an unbounded buffer in feature calls such as *buffer.put* (*n*) and *buffer.item*. To ensure exclusive access, the consumer must lock the buffer before accessing it. Such locking requirements of a feature must be expressed in the formal argument list: any target of separate type within the feature must occur as a formal argument; the arguments' handlers are locked for the duration of the feature execution, thus preventing data races. Such targets are called *controlled*. For instance, in *consume*, *buffer* is a formal argument; the consumer has exclusive access to the buffer while executing *consume*.

Condition synchronization relies on preconditions (after the **require** keyword) to express wait conditions. Any precondition makes the execution of the feature wait until the condition is true. For example, the precondition of *consume* delays the execution until the buffer is not empty. As the buffer is unbounded, the corresponding producer feature does not need a precondition.

```
consume (buffer: separate BUFFER [INTEGER])
    -- Consume an item from the buffer.
  require not (buffer.count = 0)
  local consumed_item: INTEGER
  do consumed_item := buffer.item end
```

During a feature call, the consumer could pass its locks to the buffer if it has a lock that the buffer requires. This mechanism is known as *lock passing*. In such a case, the consumer would have to wait for the passed locks to return. In *buffer.item*, the buffer does not require any locks from the consumer; hence, the consumer does not have to wait due to lock passing. However, the runtime system ensures that the result of the call *buffer.item* is properly assigned to the entity *consumed_item* using a mechanism called *wait by necessity*: while the consumer usually does not have to wait for an asynchronous call to finish, it will do so if it needs the result.

B. SCOOP Runtime

The SCOOP concepts require execution-time support, known as the SCOOP runtime. Each processor maintains a *request queue* of requests resulting from feature calls on other processors. A non-separate feature call can be processed right away without going through the request queue: the processor creates a *non-separate feature request* and processes it right away using its call stack. When the client executes a separate feature call, it enqueues a *separate feature request* to the supplier's request queue. The supplier will process the feature requests in the order of queuing.

Special attention is required in the case of *separate callbacks*, which occur for example if the buffer performs a separate feature call on the consumer, which already has a lock on the buffer. Enqueuing a feature request on the consumer could cause a deadlock if the separate callback is synchronous since the consumer might already be waiting for the buffer. The solution is to add such feature requests, corresponding to separate callbacks, ahead of all others in the request queue. This ensures that the consumer can process the feature request right away and the buffer can continue.

The runtime system includes a *scheduler*, which serves as an arbiter between processors. When a processor is ready to process a feature request in its request queue, it will only be able to proceed after the request is satisfiable. In a *synchronization step*, the processor tries to obtain the locks on the arguments' handlers in a way that the precondition holds. For this purpose, the processor sends a *locking request* to the scheduler, which stores the request in a queue and schedules satisfiable requests for application. Once the scheduler satisfies the request, the processor starts an *execution step*.

Whenever the processor is ready to let go of the obtained locks, i.e., at the end of its current feature application, it issues an unlock request to each locked processor. Each locked processor will unlock itself as soon as it processed all previous feature requests. In the example, the producer issues an unlock request to the buffer after it issued a feature request for *put*.

III. Formal Specification of SCOOP

This section outlines the formal specification for the SCOOP model. For space reasons, it only presents the main idea. The complete formal specification [17] covers all aspects of the model including arbitrarily nested feature calls, expanded types, the deep import mechanism, once routines, contract

evaluations, and explicit processor tags [13], [16]. This comprehensive coverage provides a good basis for developing compiler and runtime support; it also distinguishes this work from earlier formal specifications for SCOOP [9], [10].

A. Specifying the State

A number of abstract data types (ADTs) [18] model the state of a SCOOP program. ADTs permit a modular specification of the state on an abstract level, supporting a wide range of implementations. An ADT consists of *queries*, *commands*, and *constructors*. Each of these *features* can have a precondition that must be satisfied to call the feature; axioms describe the effects of the commands and constructors on the queries. For instance, the ADT **REG** manages the association between objects and processors: objects that are handled by the same processor form a *region*. The following two queries keep track of the available processors and the references to their objects:

$procs\colon \mathbf{REG} \rightarrow \mathbf{SET}[\mathbf{PROC}]$

$handled_objs\colon \mathbf{REG} \rightarrow \mathbf{PROC} \nrightarrow \mathbf{SET}[\mathbf{REF}]$
$\quad k.handled_objs(p)$ **require** $k.procs.has(p)$

The query *handled_objs* takes a processor of type **PROC** as an argument. The result of the first query is a set of processors with type **SET**[**PROC**]; the second query returns a set of references with type **SET**[**REF**]. The precondition of the second query makes the feature partial, indicated by $\nrightarrow$.

Besides the mapping of objects to processors, regions also keep track of locks. Developing the formal specification helped us clarifying this aspect. The informal description defines one lock per processor. However, this definition leads to a contradiction with respect to separate callbacks. A separate callback (see Section II) occurs if a processor p performs a feature call f to a processor q and q has a lock on p, as shown in Figure 1.

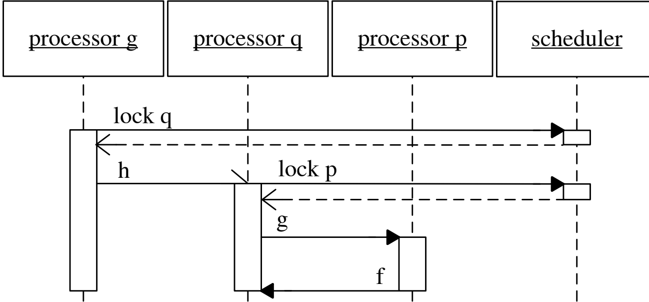

Fig. 1. A separate callback. Processor g locks q and calls asynchronously. Processor q locks p and calls synchronously. Processor p calls back synchronously.

To avoid a deadlock, p asks q to process the resulting feature request right away using its call stack. The separate callback is only possible if p has a lock on q. However, p does not have this lock because the lock is in possession of processor g that locked q. To address this design flaw, the formal specification differentiates between two types of locks: each processor has a *lock* for its *request queue* and a *lock* for its *call stack*. With this change, separate callbacks can be handled cleanly. At creation, each processor has its call stack lock. For a separate callback, processor q must pass its locks to processor p. Processor p can then use q's call stack lock to perform a separate callback to q. At that point, q's request queue lock is in possession of g, but this lock is not required by p. During the separate callback, processor p passes all its locks so that q can process the feature request. The following queries of **REG** reflect this change. The *retrieved locks* are those that got passed; the *obtained locks* are newly acquired. The locks are organized in stacks to accommodate nested feature applications:

$rq_locked\colon \mathbf{REG} \rightarrow \mathbf{PROC} \nrightarrow \mathbf{BOOLEAN}$
$\quad k.rq_locked(p)$ **require** $k.procs.has(p)$

$cs_locked\colon \mathbf{REG} \rightarrow \mathbf{PROC} \nrightarrow \mathbf{BOOLEAN}$
$\quad k.cs_locked(p)$ **require** $k.procs.has(p)$

$obtained_rq_locks\colon \mathbf{REG} \rightarrow \mathbf{PROC} \nrightarrow \mathbf{STACK}[\mathbf{SET}[\mathbf{PROC}]]$
$\quad k.obtained_rq_locks(p)$ **require** $k.procs.has(p)$

$obtained_cs_lock\colon \mathbf{REG} \rightarrow \mathbf{PROC} \nrightarrow \mathbf{PROC}$
$\quad k.obtained_cs_lock(p)$ **require** $k.procs.has(p)$

$retrieved_rq_locks\colon \mathbf{REG} \rightarrow \mathbf{PROC} \nrightarrow \mathbf{STACK}[\mathbf{SET}[\mathbf{PROC}]]$
$\quad k.retrieved_rq_locks(p)$ **require** $k.procs.has(p)$

$retrieved_cs_locks\colon \mathbf{REG} \rightarrow \mathbf{PROC} \nrightarrow \mathbf{STACK}[\mathbf{SET}[\mathbf{PROC}]]$
$\quad k.retrieved_cs_locks(p)$ **require** $k.procs.has(p)$

$locks_passed\colon \mathbf{REG} \rightarrow \mathbf{PROC} \nrightarrow \mathbf{BOOLEAN}$
$\quad k.locks_passed(p)$ **require** $k.procs.has(p)$

REG has a number of commands to add processors, add objects, lock or unlock request queues, and pass or revoke locks. The following two commands add processors and objects:

$add_proc\colon \mathbf{REG} \rightarrow \mathbf{PROC} \nrightarrow \mathbf{REG}$
$\quad k.add_proc(p)$ **require**
$\qquad \neg k.procs.has(p)$
$\quad$ **axioms**
$\qquad k.add_proc(p).procs.has(p)$
$\qquad k.add_proc(p).handled_objs(p).is_empty$
$\qquad \neg k.add_proc(p).rq_locked(p)$
$\qquad k.add_proc(p).cs_locked(p)$
$\qquad k.add_proc(p).obtained_rq_locks(p).is_empty$
$\qquad k.add_proc(p).obtained_cs_lock(p) = p$
$\qquad k.add_proc(p).retrieved_rq_locks(p).is_empty$
$\qquad k.add_proc(p).retrieved_cs_locks(p).is_empty$
$\qquad \neg k.add_proc(p).locks_passed(p)$

$add_obj\colon \mathbf{REG} \rightarrow \mathbf{PROC} \nrightarrow \mathbf{REF} \nrightarrow \mathbf{REG}$
$\quad k.add_obj(p,r)$ **require**
$\qquad k.procs.has(p)$
$\qquad \forall q \in k.procs, x \in k.handled_objs(q)\colon x.id \neq r.id$
$\quad$ **axioms**
$\qquad k.add_obj(p,r).handled_objs(p).has(r)$

REG has one constructor that returns empty regions, i.e., in the beginning, the system has no processors and no objects:

make : **REG**
  axioms *make.procs.is_empty*

The full state [17] consists of three parts: the regions, the heap, and the store. The main purpose of the heap is to keep track of objects and to maintain the mapping of references to objects; the store maintains stacks of environments that map variable names to references. Each of these parts has its own ADT along with various smaller ADTs for the basic elements of the state: objects, references, identifiers, names, processors, types, and variable environments. The consolidating **STATE** ADT offers a convenient interface to these parts. For instance, **STATE** has two features *rq_locks* and *cs_locks* that return all obtained and retrieved locks of a processor. In total, the state has 50 queries, 38 commands, and eight constructors distributed over eight ADTs; two more ADTs cover identifiers and names.

B. Specifying Executions

A structural operational semantics [19] describes the execution of a SCOOP program. A *computation* is a sequence of configurations; each *configuration* is an execution snapshot: the state, i.e., an instance of **STATE**, and the *schedule*, i.e., the call stacks and the request queues of the processors. The call stack and request queue of a processor are also known as the *action queue* of the processor. The elements in an action queue are *statements*; they can either be *instructions*, i.e., program elements, or *operations*, i.e., runtime elements. The following *transition rule*, as an example of the 41 rules in total, defines an operation that allows a processor p to issue statements s_w to a processor q during a separate callback.

Issue Operation – Separate Callback

$$\frac{\begin{array}{l} q \neq p \wedge (\sigma.rq_locks(q).has(p) \vee \sigma.cs_locks(q).has(p)) \\ \neg\sigma.locks_passed(p) \wedge \sigma.cs_locks(p).has(q) \end{array}}{\Gamma \vdash \langle p :: \texttt{issue}(q, s_w); s_p \mid q :: s_q, \sigma \rangle \rightarrow \langle p :: s_p \mid q :: s_w; s_q, \sigma \rangle}$$

The *premise* requires that processors p and q are different (first condition), q has a lock on p (first condition), and p has q's call stack lock (second condition). The first condition characterizes a separate callback; the second condition states that p needs q's call stack lock to perform the separate callback.

The bottom half of the transition rule contains the *transition definition*: the *start configuration*, the *result configuration*, and the *typing environment* Γ with the class hierarchy. The commutative and associative *processor separator* $|$ keeps the processors' action queues apart. The transition rule at hand specifies that the issue operation adds the statements s_w in front of q's action queue; the state σ remains unchanged.

The *initial configuration*, i.e., the starting point of the execution, is based on the root procedure of the program. The transition rules define how the initial configuration evolves into the *final configuration*, i.e., the end of the execution.

IV. Implementation of the Formal Specification

This section describes the implementation of the formal specification in Maude. The full executable semantics can be downloaded from [14].

A. Background on Maude

Maude [11], [12] is a programming language for theories in *membership equational logic* [20], [21] and *rewriting logic* [22], [23]. A membership equational logic is a Horn logic whose basic predicates either express membership assertions, stating that a *term* built with *operators* belongs to a certain *sort*, or equations between terms. A rewrite logic has *rewrite rules* that express conditional rewrites between terms.

A *functional module* (**fmod**) contains a membership equational theory. It defines the sorts (**sort**), subsort relationships (**subsort**), operators (**op** and **ctor** for constructors), and equations (**eq** and **ceq**) for the theory. A *system module* (**mod**) contains a rewrite theory with rewrite rules (**rl** and **crl**); it can also contain a membership equational theory. A module can be *parametrized*. To use such a module, each parameter must be bound to another module, i.e., the *parameter module*. A *theory* defines the sorts, subsort relationships, operators, equations, and rewrite rules that the parameter module must provide. A *view* specifies how the parameter module satisfies that theory.

To inspect rewrite theories, Maude offers a number of commands. The *rewrite command* takes a term and an upper bound for the number of rewrite steps; it then applies fitting rewrite rules using a rule-fair top-down strategy until it reaches the upper bound, or it finds no more fitting rewrite rules. The *frewrite command* is similar, but it uses a rule- and position-fair strategy. The *search command* explores all reachable terms until it finds a specified term.

B. Implementing ADTs

Two functional modules implement one ADT. The *ADT sort module* defines a sort that represents the ADT in Maude; it also defines subsort relationships. The *ADT module* then includes the ADT sort module to implement the ADT features with operators and equations. The two functional modules are necessary so that the ADT module itself can use instances of the ADT in parametrized collections; a single ADT module would create a circular dependency between the ADT module and the collection module. If instances of the ADT can become members of a parametrized collection, then the ADT has a view to facilitate this. **REG** from Section III-A leads to the following two functional modules; a view is not necessary for **REG**:

fmod *REGIONS−SORTS* **is sort** *Regions* . **endfm**
fmod *REGIONS* **is including** *REGIONS−SORTS* . ... **endfm**

The ADT module of an ADT defines one constructor that determines the internal structure of an ADT instance; this structure reflects the data of an ADT instance as defined by the queries. A number of variables relate to an ADT instance

and its data. For **REG**, the internal structure holds the handled objects *ho* as a map *Map*{*Proc*, *RefSet*}; this map also keeps the available processors in the key set of the map. Next, the internal structure remembers which request queues and call stacks are locked using two maps *rql* and *csl* of type *Map*{ *Proc*, *Bool*}. The next four items *orq*, *ocs*, *rrq*, and *rcs* manage the obtained and retrieved locks. Finally, *lp* remembers which processors passed locks.

```
op regions : Map{Proc, RefSet} Map{Proc, Bool} Map{
    Proc, Bool} Map{Proc, ProcSetList} Map{Proc, Proc}
    Map{Proc, ProcSetList} Map{Proc, ProcSetList} Map{
    Proc, Bool} -> Regions [ctor] .

var k : Regions . var p : Proc . var r : Ref .
var ho : Map{Proc, RefSet} .
var rql csl lp : Map{Proc, Bool} .
var orq rrq rcs : Map{Proc, ProcSetList} .
var ocs : Map{Proc, Proc} .
```

The ADT module then defines operators and equations for the ADT's features. Each query leads to one operator and one equation. The operator reflects the structure of the query: the syntax along with the sorts for the ADT instance to operate on, the formal arguments, and the result. The equation links the query to the internal structure of the ADT instance. If the query has a precondition, then the equation has a corresponding condition. The following code shows the first three queries of **REG**; for space reasons, precedence values and formatting specifications are omitted:

```
op _.procs : Regions -> Set{Proc} .
eq regions(ho, rql, csl, orq, ocs, rrq, rcs, lp)
  .procs = ho .keys .

op _.handledObjs(_) : Regions Proc -> Set{Ref} .
ceq regions(ho, rql, csl, orq, ocs, rrq, rcs, lp)
  .handledObjs (p) = ho[p] .values
  if ho .keys .has(p) .

op _.isRqLocked(_) : Regions Proc -> Bool .
eq regions(ho, rql, csl, orq, ocs, rrq, rcs, lp)
  .isRqLocked (p) = rql[p] .
  if ho .keys .has(p) .
```

Each command also leads to one operator and one equation. However, the structure of a command is different because the result of a command is an updated ADT instance. The equation reflects the axioms of the command: it defines how to rewrite a command call on an ADT instance into an updated ADT instance. The equation can define auxiliary terms in the condition. As before, the condition also contains the precondition of the command. The following code shows the two commands to add processors and objects; in there, the auxiliary operator *refIdUnique* returns whether the regions already contain a given reference or not.

```
op _.addProc(_) : Regions Proc -> Regions .
ceq k .addProc (p) =
  regions((ho .insert(p --> empty)), (rql[p] ::= false),
    (csl[p] ::= true), (orq[p] ::= nil), (ocs[p] ::= p),
    (rrq[p] ::= nil), (rcs[p] ::= nil), (lp[p] ::= false))
  if regions(ho, rql, csl, orq, ocs, rrq, rcs, lp) := k ∧
    not k .procs .has(p) .

op _.addObj(_, _) : Regions Proc Obj -> Regions .
ceq k .addObj (p, r) =
  regions((ho .insert(p --> (ho[p] U {r}))),
    rql, csl, orq, ocs, rrq, rcs, lp)
  if regions(ho, rql, csl, orq, ocs, rrq, rcs, lp) := k ∧
    k .procs .has(p) ∧ k .refIdUnique(r) .
```

A constructor is similar to a command with the difference that it does not operate on an ADT instance. The constructor of **REG** is very simple:

```
op new REGIONS.make : -> Regions .
eq new REGIONS.make =
  regions(empty, empty, empty, empty,
    empty, empty, empty, empty) .
```

With the ADT modules, Maude can reduce feature call chains. These feature call chains can be built because constructors and commands always return a new or updated ADT instance that can then be used by a subsequent command or query call to operate on.

C. Implementing Transition Rules

To implement transition rules in Maude, some basic support is necessary. A number of functional modules model a SCOOP program using sorts and operators for programs, classes, features, expressions, instructions, and types; other functional modules model operations. Instructions and operations are subsorts of statements. Views ensure that they can be stored in parametrized collections.

A new system module defines sorts and operators for action queues and configurations. A list of action queues is partitioned by the associative and commutative processor separator; it also has an identity element.

```
sorts ActionQueue ActionQueueList Configuration .
subsort ActionQueue < ActionQueueList .

op {_} _ :: _ : Nat Proc List{Statement} ->
  ActionQueue [ctor] .
op nil : -> ActionQueueList [ctor] .
op _ | _ : ActionQueueList ActionQueueList ->
  ActionQueueList [ctor assoc comm id: nil] .
op _ |- _ , _ , _ : Program ActionQueueList Nat State ->
  Configuration [ctor] .
```

By comparing this implementation to the configuration in Section III-B, one can notice three deviations. First, an action queue has an additional natural number to associate a priority to each processor; this priority is used for scheduling (see Section IV-D). Second, the configuration has an additional natural number to count the number of steps in the execution. This counter provides a continuous stream of numbers and is used to create fresh identifiers; this link between the step numbers and the identifiers helps in debugging because one can easily determine in which step Maude created an identifier. Third, each configuration has a program associated with it; this program implements the typing environment. In Section III-B, the typing environment only occurs as part of the transition definition. With this deviation, Maude does not have to add the typing environment to each configuration before executing a transition rule. With this basic support, the system module can implement transition rules. First, it defines a number of variables:

```
var p q : Proc . var qs : Set{Proc} . var σ : State .
var f : Feature . var sw sp sq : List{Statement} .
var a : Channel . var i j ic : Nat .
```

Each SCOOP operation leads to a sort, a subsort relationship, and an operator. The following code implements the issue operation from Section III-B; as before, precedence values and formatting specifications are omitted:

```
sort Issue .
subsort Issue < Operation .
op issue(_, _) : Proc List{Statement} -> Issue [ctor] .
```

Each transition rule leads to a conditional rewrite rule. The condition of the rewrite rule contains the premise of the transition rule. The label contains the name of the transition rule. The rewrite arrow => separates the start configuration from the result configuration. For instance, the transition rule for the issue operation becomes:

```
crl [issueSeparateCallback] :
  Γ |− {i} p :: issue(q, sw) ; sp | {j} q :: sq, ic, σ =>
  Γ |− {i} p :: sp | {j} q :: sw sq, ic + 1, σ
  if
    q =/= p and
      (σ .rqLocks(q).has(p) or σ .csLocks(q).has(p)) ∧
    not σ .areLocksPassed(p) and σ .csLocks(p).has(q)
```

Using the transition rules, one can then execute a SCOOP program by providing an initial configuration that captures the starting point of the program. One can then ask Maude to find one possible terminal configuration using the rewrite or frewrite command; one can also ask Maude to search for a specified configuration using the search command.

D. Scheduling

Without priorities in action queues, the rewrite command becomes problematic when a sequence of transitions leads to a result configuration that is equal to the start configuration. Maude would apply the same transition rules using the same action queues over and over again instead of continuing with other action queues. The solution takes advantage of the fact that Maude brings configurations into a canonical form before applying transition rules. During this reduction, Maude orders the action queues since the processor separator | is commutative and associative. The priorities in the action queues influence how Maude orders the action queues; consequently, the priorities also influence which action queues Maude uses next.

For example, the formal specification has a transition rule for an operation that locks a set of request queues and evaluates a precondition. If the precondition is satisfied, the operation is done; otherwise, the operation unlocks the request queues, removes the locks from the stack, and starts from scratch. Without priorities, the result configuration would be equal to the start configuration in the latter case. With priorities, the operation can decrease the priority in its action queue after determining that the precondition is not satisfied, causing Maude to focus on another action queue. The following code shows this in more detail:

```
crl [checkPreAndLock] :
  Γ |− {i} p :: checkPreAndLock(qs, f) ; sp, ic, σ =>
  Γ |− {i} p :: lock(qs) ; eval(a, f .pre) ; wait(a, p) ;
    provided not a .data then
      nissue(qs, unlockRq ;) popObtainedLocks ;
      yield ; checkPreAndLock(qs, f) ;
    end ; sp, ic + 1, σ
  if a := fresh(ic, 1) .
```

In this code, *yield* decreases the priority. The channel *a* carries the result of the precondition evaluation in *eval*, and *wait* blocks until the result is available. The *nissue* operation issues to a set of processors, and *popObtainedLocks* removes the top set from the stack of obtained request queue locks.

The priorities are not just useful to ensure progress in Maude; they are also useful to implement and test different scheduling algorithms: a scheduler can use the priorities to influence which action queues Maude uses for the next transition. The current scheduler is deterministic (just as the built-in Maude scheduler), ensuring that each execution is reproducible. To observe a variety of program executions, one can use the priorities to implement a nondeterministic scheduler that maximizes coverage or generates random schedules.

V. Testing the formal specification

This section demonstrates how testing with the executable formal specification helped to discover and fix three flaws in the SCOOP model. It describes each flaw using a simplified test program and then shows the corrections in the formal

specification. For readability, it presents the test programs in regular SCOOP syntax instead of the equivalent Maude form used for testing.

We distilled these test programs from a larger test suite [14] with (1) programs that use different combinations of SCOOP aspects and (2) programs that cover real-world scenarios (parallel searching, share market simulation, pipeline computation, producer-consumer, concurrent linked list). We fed each of these programs along with the executable formal specification to Maude. We then used the rewrite command with progressive upper bounds of rewrite steps to explore various executions. The resulting sequences of configurations, with complete schedule and state, provided enough information to identify design flaws. One behavior was particularly helpful: Maude stops when it can no longer simplify ADT feature calls or find a fitting transition rule. This can be caused by a failed precondition in an ADT feature, a failed premise of a transition rule, or a start configuration in a transition rule that does not fit.

A. Separate Callbacks

A separate callback is characterized by the following condition: at the moment of a feature call, the supplier has a lock on the client. While this condition adequately captures simple scenarios with only two processors, it fails to capture more complex scenarios as shown in the following listing:

```
class A create make feature
  b: separate B
  c: separate C
  make
    do
      create b; create c; f (b)
    end
  f (b: separate B)
    do b.g (c, Current) end
  k: separate D
    do
      create Result
    end
end

class B feature
  g (c: separate C; a: separate A)
    do c.h (a) end
end

class C feature
  h (a: separate A)
    local t: separate D
    do t := a.k end
end

class D end
```

For space reasons, this section does not show the full schedule and state; instead, it uses an abstract form of the Maude output that only shows the obtained request queue locks (orq), the retrieved request queue locks (rrq), the retrieved call stack locks (rcs), whether a processor's request queue is locked, and whether a processor passed its locks.

To start, the system creates an object of type *A* on a new root processor *a*; processor *a* then creates two objects on two new processors *b* and *c* and executes feature *f*. Maude reports accordingly:

$a ::$ orq: $(\{\},\{b\})$ rrq: $(\{\},\{\})$ rcs: $(\{\},\{\})$ locked
$b ::$ orq: $()$ rrq: $()$ rcs: $()$ locked
$c ::$ orq: $()$ rrq: $()$ rcs: $()$ unlocked

Processor *a* then calls processor *b*. The call is synchronous as it involves lock passing triggered by the reference to the current object as an actual argument. Processor *a* passes both the request queue lock on *b* and its own call stack lock. Processor *b* then obtains the request queue lock of *c*:

$a ::$ orq: $(\{\},\{b\})$ rrq: $(\{\},\{\})$ rcs: $(\{\},\{\})$ locked passed
$b ::$ orq: $(\{c\})$ rrq: $(\{b\})$ rcs: $(\{a\})$ locked
$c ::$ orq: $()$ rrq: $()$ rcs: $()$ locked

Processor *b* then passes all its locks to processor *c* during a synchronous call with lock passing:

$a ::$ orq: $(\{\},\{b\})$ rrq: $(\{\},\{\})$ rcs: $(\{\},\{\})$ locked passed
$b ::$ orq: $(\{c\})$ rrq: $(\{b\})$ rcs: $(\{a\})$ locked passed
$c ::$ orq: $(\{\})$ rrq: $(\{c,b\})$ rcs: $(\{a,b\})$ locked

Finally, processor *c* synchronously calls processor *a*, which is waiting for its synchronous call to return. However, the feature call does not qualify as a separate callback because processor *a* does not have a lock on processor *c*. Therefore, processor *c* performs a regular separate feature call and adds its feature request to the end of processor *a*'s request queue. At this point, all processors wait. Maude cannot process any of the wait operations because no processor completes its feature request. This design flaw can be resolved by fixing the separate callback condition:

Issue Operation – Separate Callback

$$\frac{q \neq p \wedge \sigma.locks_passed(q) \wedge \neg\sigma.locks_passed(p) \wedge \sigma.cs_locks(p).has(q) \quad \neg\sigma.locks_passed(p) \wedge \sigma.cs_locks(p).has(q)}{\Gamma \vdash \langle p :: \mathtt{issue}(q, s_w); s_p \mid q :: s_q, \sigma\rangle \rightarrow \langle p :: s_p \mid q :: s_w; s_q, \sigma\rangle}$$

A separate callback is now characterized as: the supplier has passed its locks, and the client has the supplier's call stack lock (first condition). This condition implies that the supplier is waiting because it passed its locks over a chain of feature calls all the way to the client. In this situation, it is necessary for the supplier to process a feature request right away; otherwise, a deadlock would occur. In addition to the simple scenarios with only two processors, this condition also captures more complex scenarios such as the one shown here. The change triggered a number of similar changes in other transition rules, not shown here for space reasons.

B. Separate Once Functions

A *once function* gets executed at most once in a context. Once functions declared as separate have a once per system semantics; non-separate once functions have a once per processor semantics. The result from the first execution in a context becomes the result of all future executions in the same context. If a once function has been executed in a context, it is *non-fresh* in that context; otherwise it is *fresh*.

Two transition rules describe how to process a feature request for a once function; the feature request comes in the form of an apply operation. The first transition rule has a condition $\sigma.is_fresh(p, f)$ and only applies to once functions f that are fresh in the context p. The apply operation immediately sets f to non-fresh with the void result. It then updates the once result during the execution. The second transition rule has a condition $\neg\sigma.is_fresh(p, f)$ and only applies to once functions f that are non-fresh in the context p. In this case, the apply operation returns the previous result on p.

This specification is problematic when a once function is of separate type, and two processors execute the once function concurrently. The following program shows this design flaw in more detail:

```
class A create make feature
  make
    local b1, b2: separate B
    do
      create b1; create b2; f (b1, b2)
    end
  f (b1, b2: separate B)
    do
      b1.g; b2.g
    ensure b1.c = b2.c
    end
end

class B feature
  c: separate C
  g
    do c := h end
  h: separate C
    once
      create Result
    end
end

class C end
```

A new root processor a creates two objects on two new processors b_1 and b_2; it then asks both processors to asynchronously execute g, causing both of them to assign the result of the separate once function h to c. One of the processors, say b_1, is first and sets h to non-fresh with the void result, as can be seen in the abstract Maude output: all :: $\{B\}.h \rightarrow void$. When processor b_2 begins, it cannot execute h again because h is already non-fresh. Consequently, b_2 just queries the current value, which is void. Processor b_1 then updates the result with a reference r to a new object: all :: $\{B\}.h \rightarrow r$. When processor a evaluates its postcondition and compares the two results of the same once function, it finds the results to be different. This finding contradicts the idea of once functions.

This design flaw can be fixed with a new status sequence for once functions: initially, a once function is fresh in a context; just after its first execution in the context, it becomes non-fresh and *not stable*; finally, it becomes non-fresh and *stable* when the first execution is over. The processor that executes the once function for the first time in a context is the *stabilizer*. With this new status sequence, a processor different from the stabilizer can now be prevented from getting the result too soon. The condition of the transition rule for non-fresh once functions becomes: $\neg\sigma.is_fresh(p, f) \wedge (\sigma.is_stable(p, f) \vee \sigma.stabilizer(p, f) = p)$. It still allows the stabilizer to recursively call the once function without blocking, in which case the result is the last computed value.

C. Lock Passing for Queries

A client only passes its locks when it performs a separate callback or when it has a lock that the supplier requires directly. Consequently, it does not pass its locks in a regular query call without arguments. This is an issue when the supplier requires one of the client's locks in a later call, as can be observed in the following code:

```
class A create make feature
  b: separate B
  c: separate C
  make
    do
      create c; create b.make (c); f (b, c)
    end

  f (b: separate B; c: separate C)
    local t: separate D
    do t := b.g; c.k end
end

class B create make feature
  w: separate C
  make (c: separate C)
    do w := c end
  g: D
    do h (w); create Result end
  h (c: separate C)
    do c.k end
end

class C feature
  k do end
end

class D end
```

A new root processor a creates two objects on two new processors b and c. It then starts executing the feature f and locks the two request queues, as can be seen in the following Maude output:

$a ::$ orq: $(\{\},\{b,c\})$ rrq: $(\{\},\{\})$ rcs: $(\{\},\{\})$ locked
$b ::$ orq: () rrq: () rcs: () locked
$c ::$ orq: () rrq: () rcs: () locked

In f, processor a performs a query call to processor b. This query call is synchronous due to wait by necessity; however, it does not involve lock passing as processor b does not require any locks from processor a to execute g:

$a ::$ orq: $(\{\},\{b,c\})$ rrq: $(\{\},\{\})$ rcs: $(\{\},\{\})$ locked
$b ::$ orq: $(\{\})$ rrq: $(\{\})$ rcs: $(\{\})$ locked
$c ::$ orq: () rrq: () rcs: () locked

Processor b then tries to obtain processor c's request queue lock as it executes h (w). However, since processor a still holds on to this lock while waiting for its query call to return, the system deadlocks. Maude stops because it cannot satisfy the premise of the lock operation.

This design flaw can be addressed by always passing locks during a query call. This is not harmful because the client must wait anyway. The small change also facilitates separate callbacks where the feature call chain involves query calls (see Section V-A).

D. Coverage

Testing concurrent programs is known to be difficult because of the nondeterminism in their execution, which causes some faults to be exposed only for certain schedules. Testing concurrency *models* on the other hand seems to be a much easier problem. The reason for this is that testing a model amounts to probing the correctness of the transition rules, and most schedules will still cause the same transition rules to be applied.

Indeed, the scheduler used in the tests is deterministic but was successful in exposing flaws in the model. When creating test programs for testing concurrency models, it is therefore important to maximize coverage of the transition rules – rather than maximizing coverage of states in the execution traces, as in the testing of concurrent programs.

VI. Related Work

Brooke, Paige, and Jacob [9] present a CSP model of a SCOOP subset and use a model checker to test the model with example programs. They explore different lock passing mechanisms and different strategies to release locks. They conclude that a supplier can unlock as soon as its client stops using it. Compared to our formal specification, which comprehensively models SCOOP, the CSP model focuses on a number of core SCOOP concepts. Ostroff et al. [10] also present a formal specification for a SCOOP subset and use it in a model checker to verify SCOOP programs. In contrast, our approach focuses on discovering flaws in the SCOOP model and not on program verification.

Ellison and Rosu [24] also combine formal specification and testing. They use K [25] to describe an executable formal specification of C and gain confidence in their specification by testing it against a GCC test suite. Their approach is similar to our approach, with the major difference that our approach deals with a concurrency model. The importance of testing executable formal specifications of programming models is also emphasized by Klein et al. [26].

Several works [27], [28] present tools to explore memory models using formal specifications and concurrent test programs. These tools exhaustively find all executions allowed by the specified memory model and then check these executions for failures. These works do not comprehensively capture a concurrency model as they focus on the memory model; hence, they are not suited to study full-fledged concurrent programs.

Verdejo and Martí-Oliet [29] describe various executable formal specifications implemented in Maude. Some of those specifications have intricate features, which make the implementation challenging, but they are still very compact compared to SCOOP. Previously, Thatia, Sen, and Martí-Oliet created an executable formal specification [30] of an asynchronous version of the π-calculus.

The maturity of reasoning frameworks has also led to a growing interest in large formal specifications of programming language semantics and the compilation process. For example, Leroy [7] uses the Coq proof assistant to program the compiler for a C-like imperative language and to prove its correctness. This approach of a fully formal proof of compilation correctness has also been applied in the context of concurrent languages. Lochbihler [8] presents a formalization of concurrent Java and proves the correctness of the source to bytecode compilation using Isabelle/HOL. Batty et al. [31] provide models for the revised standards of C and C++ which add concurrency to the languages. They are able to prove, using the theorem prover HOL4, the correctness of the proposed compilation schemes. Our approach emphasizes the value of an executable semantics in order to perform testing, which arguably gives a faster feedback on the model than fully formal proofs.

A number of semantic frameworks other than Maude enable work into large formal specifications. K [25] is a rewrite-based executable semantic framework that builds on top of Maude. Its rewrite rules generalize over traditional ones by being able to specify which parts of a term they read, write, or do not care. PLT Redex [32] is a language designed for specifying and debugging operational semantics, requiring a grammar and reduction rules as the only input. It allows to visualize reductions and to check subject reduction theorems. Ott [33] provides a metalanguage and a tool to express semantics and to compile them into code that can be interpreted by proof assistants. Since our primary focus was on deriving an executable semantics in order to facilitate testing rather than proofs, we did not consider using Ott.

VII. Conclusion

We presented a development methodology for concurrency models that relies on executable formal specifications to conduct testing prior to developing compiler and runtime support. We applied this methodology to SCOOP, resulting in a formal specification based on structural operational semantics and ADTs, implemented in Maude. We managed to find and resolve three major design flaws: an insufficient separate callback condition, faulty separate once functions, and an insufficient lock passing condition. Having successfully demonstrated the methodology on an extensive concurrency model, we believe that it can also be beneficial to developers of other models.

One of the unexpected results is that nondeterministic execution, which complicates the testing of concurrent programs, does not turn out to be a significant issue for testing concurrency models. The results reported here suggest that executable specifications, which suffer from the same criticism of principle as program tests (in both cases the approach can at best give partial reassurance, never a full guarantee), are, in practice, a realistically usable and fruitful path to the verification of semantic models.

These results open up several directions of future work: generating test programs automatically; evaluating different scheduling algorithms; running conformance tests of the SCOOP compiler and runtime system with respect to the specification; using the executable specification as a SCOOP interpreter for teaching purposes; and applying the approach to other models.

Acknowledgments

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 291389, the Hasler Foundation, and ETH (ETHIIRA).

References

[1] R. S. Carl Hewitt, Peter Bishop, "A universal modular ACTOR formalism for artificial intelligence," in *International Joint Conference on Artificial Intelligence*, 1973, pp. 235–245.

[2] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985.

[3] C. Fournet and G. Gonthier, "The reflexive CHAM and the join-calculus," in *Symposium on Principles of Programming Languages*, 1996, pp. 372–385.

[4] G. Almasi, "Partitioned global address space (PGAS) languages," in *Encyclopedia of Parallel Computing*, D. A. Padua, Ed. Springer, 2011, p. 1465.

[5] D. Gelernter, N. Carriero, S. Chandran, and S. Chang, "Parallel programming in Linda," in *International Conference on Parallel Processing*, 1985, pp. 255–263.

[6] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice-Hall, 1997.

[7] X. Leroy, "Formal certification of a compiler back-end or: programming a compiler with a proof assistant," in *Symposium on Principles of Programming Languages*, 2006, pp. 42–54.

[8] A. Lochbihler, "Verifying a compiler for Java threads," in *European conference on Programming Languages and Systems*, 2010, pp. 427–447.

[9] P. J. Brooke, R. F. Paige, and J. L. Jacob, "A CSP model of Eiffel's SCOOP," *Formal Aspects of Computing*, vol. 19, no. 4, pp. 487–512, 2007.

[10] J. S. Ostroff, F. A. Torshizi, H. F. Huang, and B. Schoeller, "Beyond contracts for concurrency," *Formal Aspects of Computing*, vol. 21, no. 4, pp. 319–346, 2008.

[11] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer, "Principles of Maude," *Electronic Notes in Theoretical Computer Science*, vol. 4, pp. 65–89, 1996.

[12] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All About Maude – A High-Performance Logical Framework*. Springer, 2007.

[13] P. Nienaltowski, "Practical framework for contract-based concurrent object-oriented programming," Ph.D. dissertation, ETH Zurich, 2007.

[14] ETH Zurich, "SCOOP executable formal specification repository," http://bitbucket.org/bmorandi/scoop-executable-formal-specification/src/, 2013.

[15] B. Morandi, S. Nanz, and B. Meyer, "Who is accountable for asynchronous exceptions?" in *Asia-Pacific Software Engineering Conference*, 2012, pp. 462–471.

[16] B. Morandi, S. S. Bauer, and B. Meyer, "SCOOP – a contract-based concurrent object-oriented programming model," in *Advanced Lectures on Software Engineering*, ser. Lecture Notes in Computer Science, P. Müller, Ed. Springer, 2010, vol. 6029, pp. 41–90.

[17] B. Morandi, S. Nanz, and B. Meyer, "A formal reference for SCOOP," in *Empirical Software Engineering and Verification*, ser. Lecture Notes in Computer Science. Springer, 2012, vol. 7007, pp. 89–157.

[18] B. Liskov and S. Zilles, "Programming with abstract data types," *ACM SIGPLAN Notices*, vol. 9, no. 4, pp. 50–59, 1974.

[19] G. D. Plotkin, "A structural approach to operational semantics," *The Journal of Logic and Algebraic Programming*, vol. 60–61, pp. 17–139, 2004.

[20] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer, "Specification and proof in membership equational logic," *Theoretical Computer Science*, vol. 236, no. 1-2, pp. 35–132, 2000.

[21] J. Meseguer, "Membership algebra as a logical framework for equational specification," in *Workshop on Algebraic Development Techniques*, 1997, pp. 18–61.

[22] ——, "Conditional rewriting logic as a unified model of concurrency," *Theoretical Computer Science*, vol. 96, pp. 73–155, 1992.

[23] R. Bruni and J. Meseguer, "Generalized rewrite theories," in *International Colloquium on Automata, Languages and Programming*, 2003, pp. 252–266.

[24] C. Ellison and G. Rosu, "An executable formal semantics of C with applications," in *Symposium on Principles of Programming Languages*, 2012, pp. 533–544.

[25] G. Roşu and T. F. Şerbănuţă, "An overview of the K semantic framework," *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[26] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler, "Run your research: on the effectiveness of lightweight mechanization," in *Symposium on Principles of Programming Languages*, 2012, pp. 285–296.

[27] E. Torlak, M. Vaziri, and J. Dolby, "MemSAT: Checking axiomatic specifications of memory models," *ACM SIGPLAN Notices*, vol. 45, pp. 341–350, 2010.

[28] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, "Mathematizing C++ concurrency," *ACM SIGPLAN Notices*, vol. 46, pp. 55–66, 2011.

[29] A. Verdejo and N. Martí-Oliet, "Executable structural operational semantics in Maude," *Journal of Logic and Algebraic Programming*, vol. 67, no. 1-2, pp. 226–293, 2006.

[30] P. Thati, K. Sen, and N. Martí-Oliet, "An executable specification of asynchronous Pi-calculus semantics and may testing in Maude 2.0," *Electronic Notes in Theoretical Computer Science*, vol. 71, pp. 261–281, 2004.

[31] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell, "Clarifying and compiling C/C++ concurrency: from C++11 to POWER," in *Symposium on Principles of Programming Languages*, 2012, pp. 509–520.

[32] J. Matthews, R. B. Findler, M. Flatt, and M. Felleisen, "A visual environment for developing context-sensitive term rewriting systems," in *Conference on Rewriting Techniques and Applications*, ser. Lecture Notes in Computer Science, vol. 3091, 2004, pp. 301–311.

[33] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša, "Ott: effective tool support for the working semanticist," in *International Conference on Functional programming*, 2007, pp. 1–12.