Concurrent Object-Oriented Programs: From Specification to Code

Emil Sekerinski

McMaster University
Department of Computing and Software
Hamilton, Ontario, Canada
emil@mcmaster.ca

Abstract. In this paper we put forward a concurrent object-oriented programming language in which concurrency is tightly integrated with objects. Concurrency is expressed by extending classes with actions and allowing methods to be guarded. Concurrency in an object may be hidden to the outside, thus allowing concurrency to be introduced in subclasses of a class hierarchy. A disciplined form of intra-object concurrency is supported. The language is formally defined by translation to action systems. Inheritance and subtyping is also considered. A theory of class refinement is presented, allowing concurrent programs to be developed from sequential specifications. Our goal is to have direct rules for verification and refinement on one hand and a practical implementation on the other hand. We briefly sketch our implementation. While the implementation relies on threads, the management of threads is hidden to the programmer.

1 Introduction

The reason for having concurrency in programs is that concurrency occurs naturally when modeling the problem domain, is to make programs more responsive, and is to exploit the potential speedup offered by multiple processors. It has been argued that objects can be naturally thought of as evolving independently and thus concurrently; objects are a natural "unit" of concurrency. Yet, current mainstream object-oriented languages treat concurrency independently of objects: typically concurrency is expressed in terms of threads that have to be created separately from objects.

In this paper we put forward a notation for writing truly concurrent object-oriented programs. Sequential object-oriented programs are expressed in terms of classes featuring attributes and methods. We keep this paradigm and extend it by augmenting classes by *actions* and adding *guards* to methods. While methods need to be invoked, actions are executed autonomously. Atomicity of attribute access is guaranteed by allowing only one method or action to be active in an object at any time. Concurrency is achieved by having active methods and actions in several objects.

We also suggest a theory for developing concurrent object-oriented programs out of sequential ones, recognizing that concurrent programs often arise from sequential specifications. Class hierarchies are commonly used to express specification-implementation relationships. We envisage continuing to do this with concurrent classes by treating concurrency in the same way an implementation issue as the choice of a data structure.

Thus we may have a class serving as a specification and subclasses of it being sequential or concurrent implementations.

For a general overview of concurrent object-oriented languages we refer to [10]. Our work shares with the πoβλ approach by Jones et al. [14, 16] the use of synchronous communication between objects and the use of objects for restricting interference. While πoβλ is defined in terms of the π calculus, a process algebra, the definition of our language is in terms of action systems. We do not directly support *early return* and *delegate* statements as πoβλ does, but we do support inheritance and subtyping. Earlier related work includes the POOL family of languages [1], where communication between concurrent objects is done by *rendezvous*. Hoare-style verification rules for a language that includes statements for sending and receiving synchronous messages are given in [2]. Here we consider instead only (synchronous) method calls, where entrance to objects is regulated by method guards.

Several approaches have emerged from extending action systems to model concurrent objects, as there is an established theory of data refinement and atomicity refinement of action systems [3, 6]. Action systems with procedures by Back and Sere [5] and Sere and Walden [23] resemble concurrent objects, except that action systems cannot be created dynamically like objects. Bosangue, Kok and Sere [8, 9] apply action systems to model dynamically created objects. Büchi and Sekerinski [11] take this further by defining inheritance and subtyping and justify the refinement rules with respect to observable traces. However, both approaches enforce strict atomicity of actions: if an action (or method) contains several method calls that may block, either all are executed or the whole action is not enabled. Thus, these approaches do not allow direct translation to efficient code. The Seuss approach of Misra [20] is also action-based but additionally considers fairness between actions. Guarded methods are distinguished from unguarded methods, with the syntactic restriction that there can be only one call to a guarded method per action and this must be the first statement. Other restrictions are that object cannot be created dynamically and there is no inheritance.

The goal of the presented work is on one hand to have a simple theory of program development and on the other hand to have an efficient and practical implementation. This paper is the result of several iterations towards this goal, starting with [4]. To test our ideas, we have developed a prototypical compiler for our language [17]. A key concept is to weaken the strict atomicity of methods and actions: when a method call transfers control to another object, the lock to the first object is released and a new activity in that object can be initiated. Section 2 introduces the language and gives some examples.

Our approach to making the theory simple is to start with a formal model of *concurrent modules* and to express all other constructs by translations into this "core". Only translations that are needed have to be applied and all formal reasoning is done in the core. The formalization is done within the Simple Theory of Types. Section 3 formalizes that core, on top of which Section 4 defines classes, objects, inheritance, subtyping, and dynamic binding and discusses verification and refinement. Section 5 extends this to concurrent objects. Section 6 sketches the implementation of the language. We conclude with a discussion of the proposed model and the kind of concurrency it leads and with observations of the limitations of the current work.

2 A Concurrent Object-Oriented Language

We start by giving the (slightly simplified) formal syntax of the language in extended BNF. The construct $a \mid b$ stands for either a or b, $[a]$ means that a is optional, and $\{a\}$ means that a can be repeated zero or more times:

$$
\begin{array}{lll}
\textit{class} & ::= & \textbf{class}\ \textit{identifier}\,[\,\textbf{inherit}\ \textit{identifier}\,]\,[\,\textbf{extend}\ \textit{identifier}\,] \\
 & & \{\,\textit{attribute} \mid \textit{initialization} \mid \textit{method} \mid \textit{action}\,\}\ \textbf{end} \\
\textit{attribute} & ::= & \textbf{attr}\ \textit{variableList} \\
\textit{initialization} & ::= & \textbf{initialization}\ (\,\textit{variableList}\,)\,\textit{statement} \\
\textit{method} & ::= & \textbf{method}\ \textit{identifier}\,(\,\textit{variableList}, \textbf{res}\ \textit{variableList}\,) \\
 & & [\,\textbf{when}\ \textit{expression}\ \textbf{do}\,]\,\textit{statement} \\
\textit{action} & ::= & \textbf{action}\ [\,\textit{identifier}\,]\,[\,\textbf{when}\ \textit{expression}\ \textbf{do}\,]\,\textit{statement} \\
\textit{statement} & ::= & \textbf{assert}\ \textit{expression} \mid \\
 & & \textit{identiferList} := \textit{expressionList} \mid \\
 & & \textit{identiferList} :\in \textit{expressionList} \mid \\
 & & \textit{identifier.identifier}\,(\,\textit{expressionList}, \textit{identifierList}\,) \mid \\
 & & \textit{identifier} := \textbf{new}\ \textit{identifier}\,(\,\textit{expressionList}\,) \mid \\
 & & \textbf{begin}\ \textit{statement}\,\{\ ;\ \textit{statement}\,\}\ \textbf{end} \mid \\
 & & \textbf{if}\ \textit{expression}\ \textbf{then}\ \textit{statement}\,[\,\textbf{else}\ \textit{statement}\,] \mid \\
 & & \textbf{while}\ \textit{expression}\ \textbf{do}\ \textit{statement} \\
 & & \textbf{var}\ \textit{variableList} \bullet \textit{statement} \\
\textit{variableList} & ::= & \textit{identifierList} : \textit{type}\,\{\,, \textit{identifierList} : \textit{type}\,\} \\
\textit{identifierList} & ::= & \textit{identifier}\,\{\,, \textit{identifier}\,\} \\
\textit{expressionList} & ::= & \textit{expression}\,\{\,, \textit{expression}\,\}
\end{array}
$$

A class is declared by giving it a name, optionally stating the class being inherited or extended, and then listing all the attributes, initializations, methods, and actions. Initializations have only value parameters, methods may have both value and result parameters, and actions don't have parameters. Both methods and actions may optionally have a *guard*, a boolean expression. Actions may be named, though the name does not carry any meaning. The assertion statement **assert** b checks whether boolean expression b holds. If it holds, it continues, otherwise it aborts. The assignment $x := e$ assigns simultaneously the values of the list e to the list x of variables. The nondeterministic assignment statement $x :\in s$ selects an element of the set s and assigns it to the list x of variables. This statement is not part of the programming language, but is included here for use in abstract programs. A method call $o.m(e,z)$ to object o takes the list e as the value parameters and assigns the result to the list z of variables. The object creation $o :=$ **new** $C(e)$ creates a new object of class C and calls its initialization with value parameter e. We do not further define *identifier* and *expression*.

We illustrate the constructs of the language by a series of examples. Consider the problem of ensuring mutual exclusion of multiple users accessing two shared resources. A user can perform a critical section *cs* only if that user has exclusive access to both resources. We assume each resource is protected by a semaphore. Semaphores and users are represented by objects, with a semaphore having a guarded method:

```
class $Semaphore$
  attr $n : integer$
  initialization $n := 1$
  method $P$
    when $n > 0$ do $n := n - 1$
  method $V$
    $n := n + 1$
end
```

```
class $User$
  attr $s, t : Semaphore$
  initialization $(a, b : Semaphore)$
    $s, t := a, b$
  method $doCriticalSection$
    begin $s.P$ ; $t.P$ ; $cs$ ; $s.V$ ; $t.V$ end
end
```

We assume that all statements only access and update attributes of the object itself and local variables, except for method calls that may access and update the state of other objects. All statements are executed atomically up to method calls. Thus in class *Semaphore* the method V is always executed atomically, as is the initialization. The method P may block if the guard is not true, but once the method is enabled, it is also executed atomically. The method *doCriticalSection* may block at the calls $s.P$ and $t.P$. In this case some other activity must first call the V method of the corresponding semaphore before execution can resume.

The next example is about merging the elements of two bounded buffers into a third buffer. Buffers and mergers are represented by objects:

```
class $Buffer$
  attr $b$ : array of $Object$
  attr $in, out, n, max : integer$
  initialization $(m : integer)$
    $in, out, n, max := 0, 0, 0, m$ ; $b :=$ new $Object[m]$
  method $put(x : Object)$
    when $n < max$ do $in, b[in], n := (in + 1)$ mod $max, x, n + 1$
  method $get($res $x : Object)$
    when $n > 0$ do $out, x, n := (out + 1)$ mod $max, b[out], n - 1$
end

class $Merger$
  attr $in1, in2, out : Buffer$
  attr $a1, a2 : boolean$
  attr $x1, x2 : Object$
  initialization $(i1, i2, o : Buffer)$
    $in1, in2, out, a1, a2 := i1, i2, o, false, false$
  action $copy1$
    when $a1$ do begin $a1 := false$ ; $in1.get(x1)$ ; $out.put(x1)$ ; $a1 := true$ end
  action $copy2$
    when $a2$ do begin $a2 := false$ ; $in2.get(x2)$ ; $out.put(x2)$ ; $a2 := true$ end
end
```

After creating a new merger object m, the actions of m can execute in parallel with the remaining program, including other *Merger* objects. Actions cannot be called, they can be initiated automatically whenever they are enabled. Action *copy*1 is enabled if $a1$ is true. Once *copy*1 is initiated, it may block at either the call $in1.get(x1)$ or the call *out.put*$(x1)$. In this case another activity in the same object may be initiated or may resume (if it was blocked). Initiating an activity here means starting either *copy*1 or *copy*2 again. Since $a1$ is *false* at these points, *copy*1 is disabled and cannot be initiated a second time. On the other hand, *copy*2 may be initiated and come to conclusion or block at the call *in*2.*get*$(x2)$ or the call *out.put*$(x2)$. Hence for example the situation may arise that both actions are blocked at the *out.put* calls. Thus *Merger* can buffer two elements.

The last example is the observer design pattern, expressed as an abstract program. The pattern allows that all observers of one subject perform their *update* methods in parallel:

```
class Observer
   attr sub : Subject
   initialization (s : Subject)
      begin sub := s ; s.attach(this) end
   method update ...
end

class Subject
   attr obs, notifyObs : set of Observer
   initialization
      obs, notifyObs := {}, {}
   method attach(o : Observer)
      obs := obs ∪ {o}
   method notify
      notifyObs := obs
   action notifyOneObserver
      when notifyObs ≠ {} do
         var o : Observers •
         begin o :∈ notifyObs ; notifyObs := notifyObs − {o} ; o.update end
end
```

As soon as execution of the action *notifyOneObserver* in a subject s reaches the call *o.update*, control is passed to object o and another activity in s may be initiated or may resume. In particular, the action *notifyOneObserver* may be initiated again, as long as *notifyObs* is not empty, i.e. some observers have not been notified. Thus at most as many *notifyOneObserver* actions are initiated as there are observers and all notified observers can proceed concurrently. New observers can be added at any time and will be updated after the next call to *notify*.

3 Statements, Procedures, Modules, and Concurrency

We introduce the “core” language into which the object-oriented constructs are translated. The definition is done in terms of higher order logic, as the type system of higher order logic is close to that of Pascal-like languages. We assume there are some basic types like *boolean* and *integer*. New types can be constructed as functions $X \to Y$ and products $X \times Y$, for given types X and Y. Function application is written as $f(x)$ or simply $f\,x$ and a pair as (x,y) or simply x,y. For convenience we also assume that further type constructors like **set of** T and **bag of** T are available.

Statements The core statements are as follows. Let X be the type of the program state and $p : X \to boolean$ be state predicate. The assertion $\{p\}$ does nothing if p is true and aborts otherwise. The guard $[p]$ does nothing if p holds and blocks otherwise. If S and T are statements then $S\ ;\ T$ is their sequential composition. The choice $S \sqcap T$ selects either S or T nondeterministically. If Q is a relation, i.e. a function of type $X \to Y \to boolean$, then $[Q]$ is a statement that updates the state according to relation Q, choosing one state nondeterministically if several final states are possible and blocking it no final state according to Q exists. All further statements are defined in terms of these five core statements. These five statements can for example be defined by higher order predicate transformers, i.e. function mapping predicates (the postconditions) to predicates (the preconditions) as done by Back and von Wright [7].

States are typically tuples and program variables are used to selects components of the state tuple. For example, if the state space is $X = integer \times integer$ and variables x,y are used to refer to the two integer components, then a state predicate p can be defined as $p(x,y) = (x > y)$. We assume the state space is understood from context, allowing us to write boolean expressions instead of state predicates in assertions and guards, for example the assertion $\{x > y\}$.

We define $skip = \{true\} = [true]$ to be the statement that does nothing, $abort = \{false\}$ to be the statement that always aborts, and $wait = [false]$ to be the statement that always blocks. Assume b is a boolean expression. The assertion statement **assert** b is synonymous to $\{b\}$. The guarded statement **when** b **do** S and the conditional statements **if** b **then** S and **if** b **then** S **else** T are defined as:

$$\begin{array}{lcl}
\textbf{when}\ b\ \textbf{do}\ S & \widehat{=} & [b]\ ;\ S \\
\textbf{if}\ b\ \textbf{then}\ S & \widehat{=} & ([b]\ ;\ S) \sqcap [\neg b] \\
\textbf{if}\ b\ \textbf{then}\ S\ \textbf{else}\ T & \widehat{=} & ([b]\ ;\ S) \sqcap ([\neg b]\ ;\ T)
\end{array}$$

Suppose $x : X$ and $y : Y$ are the only program variables. The assignment statement $x := e$ updates x and leaves y unchanged. The nondeterministic assignment statement $x :\in s$ assigns x an arbitrary element of the set s and leaves y unchanged. If s is the empty set then the statement blocks. Both are defined in terms of an update statement:

$$\begin{array}{lclll}
x := e & \widehat{=} & [Q] & \text{where} & Q(x,y)(x',y') = (x' = e) \wedge (y' = y) \\
x :\in s & \widehat{=} & [Q] & \text{where} & Q(x,y)(x',y') = (x' \in s) \wedge (y' = y)
\end{array}$$

The declaration of a local variable **var** $x :\in s \bullet S$ extends the state space by x, executes S, and reduces the state space again. The initial value of x is chosen nondeterministically from the set s. If s is the empty set then the statement blocks. We write **var** $x : X \bullet S$ or simply **var** $x \bullet S$ if an arbitrary element of type X is chosen initially.

$$\begin{array}{lclll}
\textbf{var}\ x :\in s \bullet S & \widehat{=} & [Q]\ ;\ S\ ;\ [R] & \text{where} & Q\,y\,(x',y') = (x' \in s) \wedge (y' = y) \\
 & & & & R\,(x,y)\,y' = (y' = y)
\end{array}$$

Following theorem gives laws for transforming statements into equivalent ones. We let $e[x \backslash f]$ stand for simultaneously substituting variables x by expressions f in e:

Theorem (Equational Laws). *Assume x, y are disjoint lists of variables.*

$$x := e = x :\in \{x' \mid x' = e\} \tag{1}$$

$$x :\in \{x' \mid b\} \,;\, y :\in \{y' \mid c\} = x,y :\in \{x',y' \mid b \wedge c[x \backslash x']\} \tag{2}$$

$$\mathbf{var}\, x \bullet x,y :\in \{x',y' \mid b\} = y :\in \{y' \mid \exists x,x' \bullet b\} \tag{3}$$

For a statement S and predicate b, we let $wp(S,b)$ be the weakest precondition for S to terminate and to establish postcondition b. The *enabledness domain* or *guard* of statement S is defined by $grd\,S = \neg wp(S,false)$ and the *termination domain* by $trm\,S = wp(S,true)$. The weakest liberal precondition $wlp(S,b)$ is the weakest precondition for S to establish b provided S terminates. We give selected laws:

Theorem (Weakest Preconditions).

$$wlp(x := e, b) \;=\; b[x \backslash e] \tag{4}$$

$$wlp(x :\in s, b) \;=\; \forall x \in s \bullet b \tag{5}$$

$$wlp(S \,;\, T, b) \Leftarrow wlp(S, wlp(T,b)) \tag{6}$$

The *refinement* of statement S by T, written $S \sqsubseteq T$, means that T terminates whenever S does, T is disabled whenever S is, and T is "more deterministic" than S. In the predicate transformer model, $S \sqsubseteq T$ holds if for any postcondition q, whenever S establishes q so does T. *Data refinement* $S \sqsubseteq_R T$ generalizes (algorithmic) refinement by relating the initial and final state of S and T with relation R. We allow R to refine only part of the state, i.e. if the (initial and final) state space of S is $X \times Z$, the state space of T is $Y \times Z$, then it is sufficient for R to relate X to Y. We write Id for the identity relation and $\times$ for the parallel composition of relations:

$$S \sqsubseteq_R T \;\widehat{=}\; S \,;\, [R \times Id] \sqsubseteq [R \times Id] \,;\, T$$

We give selected laws about data refining statements; they naturally generalize when only a specific component of a larger state space is refined.

Theorem (Refinement Laws). *Assume that relation R relates X to Y and the state space includes Z. Variables x, y, z refer to the corresponding state components:*

$$x := e \sqsubseteq_R y := f \quad \textit{if} \quad Rxy \Rightarrow Ref \tag{7}$$

$$\{a\} \,;\, x := e \sqsubseteq_R \{b\} \,;\, y := f \quad \textit{iff} \quad a \wedge Rxy \Rightarrow b \quad \textit{and} \quad a \wedge Rxy \Rightarrow Ref \tag{8}$$

$$x := e \sqsubseteq_R y :\in \{y' \mid d\} \quad \textit{if} \quad Rxy \wedge d \Rightarrow Rey' \tag{9}$$

$$\{a\} \,;\, x := e \sqsubseteq_R \{b\} \,;\, y :\in \{y' \mid d\} \quad \textit{iff} \quad a \wedge Rxy \Rightarrow b \quad \textit{and} \quad a \wedge Rxy \wedge d \Rightarrow Rey' \tag{10}$$

$$x :\in \{x' \mid c\} \sqsubseteq_R y :\in \{y' \mid d\} \quad \textit{if} \quad Rxy \wedge d \Rightarrow \exists x' \bullet c \wedge Rx'y' \tag{11}$$

$$\{a\} \,;\, x :\in \{x' \mid c\} \sqsubseteq_R \{b\} \,;\, y :\in \{y' \mid d\} \quad \textit{iff} \quad a \wedge Rxy \Rightarrow b \quad \textit{and} \quad a \wedge Rxy \wedge d \Rightarrow \exists x' \bullet c \wedge Rx'y' \tag{12}$$

$$z :\in \{z' \mid c\} \sqsubseteq_R z :\in \{z' \mid d\} \quad \textit{if} \quad Rxy \wedge d \Rightarrow c \tag{13}$$

$$\{a\} \,;\, z :\in \{z' \mid c\} \sqsubseteq_R \{b\} \,;\, z :\in \{z' \mid d\} \quad \textit{if} \quad a \wedge Rxy \Rightarrow b \quad \textit{and} \quad a \wedge Rxy \wedge d \Rightarrow c \tag{14}$$

$$S_1 \,;\, S_2 \sqsubseteq_R T_1 \,;\, T_2 \quad \textit{if} \quad S_1 \sqsubseteq_R T_1 \quad \textit{and} \quad S_2 \sqsubseteq_R T_2 \tag{15}$$

$$S_1 \sqcap S_2 \sqsubseteq_R T_1 \sqcap T_2 \quad \textit{if} \quad S_1 \sqsubseteq_R T_1 \quad \textit{and} \quad S_2 \sqsubseteq_R T_2 \tag{16}$$

The iteration statement S^{ω} repeats S an arbitrary number of times, as long as S is enabled. If S never becomes disabled, then S^{ω} aborts. Iteration S^{ω} is defined as the least fixed point (with respect to the refinement relation) of the equation $X = (S \,;\, X) \sqcap \mathit{skip}$. The while statement **while** b **do** S is defined in terms of iteration, with the additional restriction that upon termination $\neg b$ must hold:

$$\textbf{while}\ b\ \textbf{do}\ S \ \ \widehat{=}\ \ ([b] \,;\, S)^{\omega} \,;\, [\neg b]$$

Modules. A module declares a number of variables with initial values as well as a number of procedures. The procedures operate on the local variables and possibly variables declared in other modules either directly or by calling other procedures. Formally a module is a pair $(\mathit{init}, \mathit{proc})$ where *init* is the initial local state and *proc* is a tuple of statements. The syntax for defining a module with a two variables p, q of types P, Q with initial values p_0, q_0 and a single procedure m is as follows:

```
module $K$
  var $p : P := p_0$
  var $q : Q := q_0$
  procedure $m(u : U,$ res $v : V)$
    $M$
end
```

Formally we have $K = (\mathit{init}, \mathit{proc})$ with $\mathit{init} = (p_0, q_0)$ and $\mathit{proc} = M$. The (initial and final) state space of the body M of m is $U \times V \times X$, where X is the state space of the whole program, which includes P and Q as components. Again, $K.p$ or simply p is the name used to select the corresponding state component. Procedure names are used for selecting the components of *proc*: we write $K.m$ or simply m in order to refer to statement M. A procedure call $m(e, z)$ extends the state space by the formal value and result parameters, copies the actual value parameters to the formal parameters, executes the procedure body, and copies the formal result parameters to the actual result parameters:

$$m(e, x) \ \ \widehat{=}\ \ \textbf{var}\ u, v \bullet u := e \,;\, m \,;\, x := v$$

Within modules other modules may be referred. The state space of the whole program is the combined state space of all modules of that program.

Concurrency. Concurrency is introduced by adding actions to modules. These actions may access variables of that module and variables of other modules, either directly or through procedures. Actions that access disjoint sets of variables may be executed in any order or in parallel. Module actions are executed atomically, i.e. either an action is enabled and can be carried to completion or it is not enabled (in contrast to class actions that are atomic only up to method calls). Formally a concurrent module is a triple $(\mathit{init}, \mathit{proc}, \mathit{act})$ where in addition *act* is the combined action of the module. We use following syntax for defining a module with actions a and b:

```
module $K$
  var $p : P := p_0$
  var $q : Q := q_0$
  procedure $m(u : U,$ res $v : V)$
    $M$
  action $a$
    $A$
  action $b$
    $B$
end
```

We have $K = (init, proc, act)$ with $init = (p_0, q_0)$, $proc = M$, and $act = A \sqcap B$. All actions are combined into a single action and the names of the actions do not carry any meaning. The state space of *act* is the state space of the whole program, which includes P and Q as components.

Definition (Module Refinement). *Module $K = (init, proc)$ with variables p is refined by module $K' = (init', proc', act)$ with variables p' through relation R, written $K \sqsubseteq_R K'$, if:*

(a) for the initialization:	$R\ init\ init'$
(b) for every procedure m:	
(b.1) procedure refinement:	$K.m \sqsubseteq_R K'.m$
(b.2) procedure enabledness:	$grd\ K.m \wedge R\ p\ p' \Rightarrow grd\ K'.m \vee grd\ act$
(c) for the action:	
(c.1) action refinement:	$skip \sqsubseteq_R act$
(c.2) action termination:	$R\ p\ p' \Rightarrow trm(\mathbf{do}\ act\ \mathbf{od})$

The loop **do** S **od** repeats S as long as it is enabled. It is defined as $S^\omega\ ;\ [\neg grd\ S]$. Compared to the **while** loop, the guard is implicit in the body. Condition (a) requires that the initializations are in the refinement relation. Condition (b.1) requires that each procedure of K is refined by the corresponding procedure of K'. While refinement by itself allows the guard to be weakened, condition (b.2) requires that whenever $K.m$ is enabled, either $K'.m$ or *act* must be enabled. Condition (c.1) requires that the effect of the action *act* is not visible when viewed from K. Finally, condition (c.2) requires that *act* eventually disables itself, hence cannot introduce non-termination.

The definition can be applied when K' has no action by taking $act = wait$. As $grd\ wait = false$ condition (b.2) simplifies to $grd\ K.m \wedge r \Rightarrow grd\ K'.m$ and condition (c) holds by default.

Module refinement can be generalized in several ways: K may also be allowed to have an action, allowing to increase the concurrency of an already concurrent module [23]. Both K and K' can exhibit finite stuttering and the generalized rule can be shown to be correct with respect to trace refinement [11]. We have restricted the refinement relation to relate only the local variables. The refinement relation can be generalized to include global variables, at the expense of losing compositionality [11, 23].

4 Objects

We distinguish between the class and the type of an object. The class defines the attributes and the methods of objects. We define a class in terms of a module with one variable for each attribute, one procedure for each method, and an extra variable for the objects populating that class. The variables map each object of the class to the corresponding attribute values. Each procedure takes an additional value parameter, *this*, for the object to which the procedure is applied. We assume the type *Object* is infinite and contains the distinguished element *nil*. All objects are of type *Object*. We write $x :\notin s$ as a shorthand for $x :\in \overline{s}$:

```
class C                          ^=  module C
  attr p : P                           var C : set of Object := {}
  initialization (g : G)               var p : Object → P
    I                                  procedure new(g : G, res this : Object)
  method l(s : S, res t : T)             this :∉ C ∪ {nil} ; C := C ∪ {this} ; I
    L                                  procedure l(this : Object, s : S, res t : T)
  method m(u : U, res v : V)             {this ∈ C} ; L
    M                                  procedure m(this : Object, u : U, res v : V)
end                                      {this ∈ C} ; M
                                     end
```

Within a method body attribute *p* is referred to by *this.p*. In general, referencing *x.p* amounts to applying the function *p* to *x*. Creating a new object *x* of class *C* with initialization parameter *e* amounts to calling the *new* procedure of class *C*. Calling the method *m* of an object *x* of class *C* amounts to calling the procedure *m* of class *C* with *x* as the additional parameter that is bound to *this* in *m*:

$$\begin{array}{lcl} x.p & \widehat{=} & p(x) \\ x := \textbf{new}\ C(e) & \widehat{=} & C.new(e,x) \\ x.m(f,z) & \widehat{=} & C.m(x,f,z) \end{array}$$

We follow the practice of using class names as if they were types in variable declarations, e.g. $c : C$. While the type of *c* is *Object*, the class name *C* is used to determine the module to which method calls to *c* go. The class name can also be used by the compiler to forbid certain assignments. We illustrate these concepts by an example of points in a plane.

```
class Point
  attr x : integer
  attr y : integer
  initialization (x : integer, y : integer)
    this.x, this.y := abs(x), abs(y)
  method distance(p : Point, res d : integer)
    d := abs(this.x − p.x) + abs(this.y − p.y)
  method copy( res p : Point)
    p := new Point(this.x + 2, this.y + 2)
end
```

Class *Point* translates to following module. We write $f[a \leftarrow b]$ for modifying function f to return b for argument a. The assignment $x.p := e$, or equivalently $p(x) := e$, stands for $p := p[x \leftarrow e]$. For convenience we continue to write $x.p$ instead of $p(x)$:

```
module $Point$
  var $Point$ : set of $Object := \{\}$
  var $x : Object \rightarrow integer$
  var $y : Object \rightarrow integer$
  procedure $new(x : integer, y : integer,$ res $this : Object)$
    $this :\notin Point \cup \{nil\}$ ; $Point := Point \cup \{this\}$ ;
    $this.x, this.y := abs(x), abs(y)$
  procedure $distance(this : Object, p : Object,$ res $d : integer)$
    $\{this \in Point\}$ ; $d := abs(this.x - p.x) + abs(this.y - p.y)$
  procedure $copy(this : Object,$ res $p : Point)$
    $\{this \in Point\}$ ; $new(this.x + 2, this.y + 2, p)$
end
```

We sketch how to verify invariance properties of classes. For example, consider showing that $(this.x \geq 0) \wedge (this.y \geq 0)$ is an invariant of class *Point*: this requires proving that I defined as $\forall this \in Point \bullet (x(this) \geq 0) \wedge (y(this) \geq 0)$ is an invariant of the module *Point*. This holds if the initial values imply the invariant, $(Point = \{\}) \Rightarrow I$, and each procedure preserves the invariant, $I \Rightarrow wlp(Point.new, I)$, $I \Rightarrow wlp(Point.distance, I)$, and $I \Rightarrow wlp(Point.copy, I)$. For *new* we have by using (4), (5), and (6):

$$
\begin{aligned}
& wlp(Point.new, I) \\
= \; & wlp(this :\notin Point \cup \{nil\} \; ; \; Point := Point \cup \{this\} \; ; \\
& \quad x(this), y(this) := abs(x), abs(y), \forall p \in Point \bullet (x(p) \geq 0) \wedge (y(p) \geq 0)) \\
\Leftarrow \; & wlp(this :\notin Point \cup \{nil\} \; ; \; Point := Point \cup \{this\}, \\
& \quad \forall p \in Point \bullet (x[this \leftarrow abs(x)](p) \geq 0) \wedge (y[this \leftarrow abs(y)](p) \geq 0)) \\
\Leftarrow \; & \forall this :\notin Point \cup \{nil\} \bullet \forall p \in Point \cup \{this\} \bullet \\
& \quad (x[this \leftarrow abs(x)](p) \geq 0) \wedge (y[this \leftarrow abs(y)](p) \geq 0)) \\
\Leftarrow \; & I
\end{aligned}
$$

While we allow references *this.a* to attributes of the object itself to be abbreviated by a, care has to be taken as this involves a hidden function application, which is the source of *aliasing*. For example, consider adding method *tile* to class *Point*:

```
method $tile(p : Point)$
  $p.x := x + 2$ ; $p.y := y$
```

We might be tempted to conclude that the postcondition $p.x = x + 2$ is always established. Expanding the body to $x(p) := x(this) + 2$; $y(p) := y(this)$ and the postcondition to $x(p) = x(this) + 2$ makes it evident that this is only true if initially $this \neq p$, i.e. the postcondition does not hold for the call $p.tile(p)$, with $p \in Point$.

We turn our attention to inheritance. Suppose C is as earlier and class D inherits from C, while adding attributes and methods and redefining the initialization and some methods. We call C the *superclass* of D and D the *subclass* of C. This corresponds to defining a module D that uses module C:

```
**class** $D$ **inherit** $C$                        $\widehat{=}$   **module** $D$
    **attr** $q : Q$                                         **var** $D :$ **set of** $Object := \{\}$
    **initialization** $(h : H)$                             **var** $q : Object \rightarrow Q$
        $J$                                                  **procedure** $new(h : H,$ **res** $this : Object)$
    **method** $m(u : U,$ **res** $v : V)$                       $this :\notin C \cup \{nil\}$ ; $C := C \cup \{this\}$ ;
        $M'$                                                     $D := D \cup \{this\}$ ; $J$
    **method** $n(w : W,$ **res** $y : Y)$                   **procedure** $l(this : Object, s : S,$ **res** $t : T)$
        $N$                                                      $\{this \in D\}$ ; $C.l(this, r, s)$
**end**                                                      **procedure** $m(this : Object, u : U,$ **res** $v : V)$
                                                                 $\{this \in D\}$ ; $M'$
                                                             **procedure** $n(this : Object, w : W,$ **res** $y : Y)$
                                                                 $\{this \in D\}$ ; $N$
                                                         **end**
```

Those methods that are not explicitly redefined in D are defined in D as forwarding the call to C. Method bodies may contain calls to other methods of the same class, either to the same object, $this.m(e,z)$ or to another object, $x.m(e,z)$. The call $this.m(e,z)$ is also written $m(e,z)$. A method body in D may also contain a super-call $super.m(e,z)$. In this case the call goes to the inherited class, i.e. the immediate superclass. This also applies to inheritance hierarchies with more than two classes:

$$super.m(e,z) \ \widehat{=}\ C.m(e,z)$$

We illustrate these issues with classes $Point1D$ and $Point2D$:

```
**class** $Point1D$                                  **class** $Point2D$ **inherit** $Point1D$
    **attr** $x : integer$                               **attr** $y : integer$
    **method** $setX(x : integer)$                       **method** $setY(y : integer)$
        $this.x := x$                                        $this.y := y$
    **method** $scale(s : integer)$                      **method** $setXY(x, y : integer)$
        $this.x := this.x \times s$                          $this.setX(x)$ ; $this.setY(y)$
**end**                                                  **method** $scale(s : integer)$
                                                             $super.scale(s)$ ; $this.y := this.y \times s$
                                                     **end**
```

These classes translate to following modules:

```
**module** $Point1D$
    **var** $Point :$ **set of** $Object := \{\}$
    **var** $x : Object \rightarrow integer$
    **procedure** $new($**res** $this : Object)$
        $this :\notin Point1D \cup \{nil\}$ ; $Point1D := Point1D \cup \{this\}$
    **procedure** $setX(this : Object, x : integer)$
        $\{this \in Point1D\}$ ; $this.x := x$
    **procedure** $scale(this : Object, s : integer)$
        $\{this \in Point1D\}$ ; $this.x := this.x \times s$
**end**
```

```
**module** $Point2D$
  **var** $Point2D : \textbf{set of } Object := \{\}$
  **var** $y : Object \rightarrow integer$
  **procedure** $new(\textbf{ res } this : Object)$
    $this :\notin Point2D \cup \{nil\}$ ; $Point1D := Point1D \cup \{this\}$ ;
    $Point2D := Point2D \cup \{this\}$
  **procedure** $setX(this : Object, x : integer)$
    $\{this \in Point2D\}$ ; $Point1D.setX(this, x)$
  **procedure** $setY(this : Object, y : integer)$
    $\{this \in Point2D\}$ ; $this.y := y$
  **procedure** $setXY(this : Object, y : integer)$
    $\{this \in Point2D\}$ ; $setX(this, x)$ ; $setY(this, y)$
  **procedure** $scale(this : Object, s : integer)$
    $\{this \in Point2D\}$ ; $Point1D.scale(this, s)$ ; $this.y := this.y \times s$
**end**
```

Inheritance does not affect the creation of objects, i.e. if D inherits from C then $x :=$ **new** $D(e) = D.new(e,x)$. A key point of the definition of inheritance is that a new object of class D becomes also a member of class C, that is D is a *subtype* of C. Subtypes correspond to subsets between the members of the class, $C \subseteq D$. Assuming c, d are objects, the type test c **is** D tests whether c is indeed an object of class D. The type cast $d := c$ **as** D aborts if c is not an object of class D and assigns c to d otherwise. Assuming D is a subtype of C and c is declared to be of class C, the method call $c.m(e,z)$ is bound dynamically, i.e. the actual class of c rather than the declared class determines the module to which the call goes. This generalizes to class hierarchies involving more than two classes accordingly:

$$\begin{array}{lcl}
c \textbf{ is } D & \widehat{=} & c \in D \\
d := c \textbf{ as } D & \widehat{=} & \{c \in D\} \; ; \; d := c \\
c.m(e,z) & \widehat{=} & \textbf{if } c \in D \textbf{ then } D.m(c,e,z) \textbf{ else } C.m(c,e,z)
\end{array}$$

Within the bodies of the methods of class D attributes of class C may be referred to. The type system would either allow or forbid this according to visibility declarations; we do not explicitly indicate visibility here. However, we note that if modification of C attributes is allowed in D, then an invariant shown to hold for C objects does not necessarily hold for D objects. Such an invariant has also to be shown to be preserved by D methods.

We can also define inheritance without subtyping, which we call *extension*. When class E extends class C, the methods of E may refer to the attributes of C, but creating an E object does not make it a C object. Methods are not inherited and no super-calls are possible (although one could generalize this). Hence this only allows sharing of attribute declarations. In case of extension the type system would forbid assignments between E and C objects:

```
class $E$ extend $C$                          $\widehat{=}$   module $E$
    attr $q : Q$                                       var $E$ : set of $Object := \{\}$
    initialization $(h : H)$                           var $q : Object \to Q$
        $J$                                            procedure $new(h : H,$ res $this : Object)$
    method $m(v : U,$ res $v : V)$                        $this :\notin D \cup \{nil\}$ ; $D := D \cup \{this\}$ ; $J$
        $M'$                                           procedure $m(this : Object, u : U,$ res $v : V)$
    method $n(w : W,$ res $y : Y)$                        $\{this \in D\}$ ; $M'$
        $N$                                            procedure $n(this : Object, w : W,$ res $y : Y)$
end                                                        $\{this \in D\}$ ; $N$
                                                   end
```

Now we show how class refinement translates to module refinement. We give an example that involves creation of auxiliary objects and creation of garbage—objects to which there is no reference. Consider following class S for defining a store in which we only record whether the store is empty or full:

```
class $S$                                  $=$   module $S$
    attr $f : boolean$                            var $S$ : set of $Object := \{\}$
    initialization                                var $f : Object \to boolean$
        $f := false$                              procedure $new($ res $this : Object)$
    method $full($ res $r : boolean)$                 $this :\notin S \cup \{nil\}$ ; $S := S \cup \{this\}$ ;
        $r := f$                                      $this.f := false$
    method $store$                                procedure $full(this : Object,$ res $r : boolean)$
        $f := true$                                   $\{this \in S\}$ ; $r := this.f$
end                                               procedure $store(this : Object)$
                                                      $\{this \in S\}$ ; $this.f := true$
                                              end
```

In the refinement LS the boolean attribute f becomes a link l to another object of class LS. Initially l is nil and is set to some object of class LS in $store$. Hence, repeated calls to $store$ will generate garbage:

```
class $LS$                                 $=$   module $LS$
    attr $l : LS$                                 var $LS$ : set of $Object := \{\}$
    initialization                                var $l : Object \to Object$
        $f := nil$                                procedure $new($ res $this : Object)$
    method $full($ res $r : boolean)$                 $this :\notin LS \cup \{nil\}$ ; $LS := LS \cup \{this\}$ ;
        $r := l \neq nil$                             $this.l := nil$
    method $store$                                procedure $full(this : Object,$ res $r : boolean)$
        $l :=$ new $LS$                               $\{this \in LS\}$ ; $r := this.l \neq nil$
end                                               procedure $store(this : Object)$
                                                      $\{this \in LS\}$ ; $new(this.l)$
                                              end
```

We show refinement between modules S and LS with relation R defined by:

$$R(S,f)(LS,l) = (S \subseteq LS) \wedge (\forall s \in S \bullet f(s) = (l(s) \neq nil))$$

Condition (a) of module refinement, $R(\{\},f)(\{\},l)$, holds immediately. To show condition (b.2) for *new* we rewrite the bodies using (1) and (2):

$$\begin{aligned}
S.new &= this,S,f := \{this',S',f' \mid \\
&\qquad (this' \notin S \cup \{nil\}) \wedge (S' = S \cup \{this'\}) \wedge (f' = f[this' \leftarrow false])\} \\
LS.new &= this,LS,l := \{this',LS',l' \mid \\
&\qquad this' \notin LS \cup \{nil\}) \wedge (LS' = LS \cup \{this'\}) \wedge (l' = l[this' \leftarrow nil])\}
\end{aligned}$$

Refinement is now established by first applying (11) and then eliminating LS',l',S',f' by the one-point rule:

$$\begin{aligned}
&S.new \sqsubseteq_R LS.new \\
= &(S \subseteq LS) \wedge (\forall s \in S \bullet f(s) = (l(s) \neq nil)) \wedge (this' \notin LS \cup \{nil\}) \wedge \\
&(LS' = LS \cup \{this'\}) \wedge (l' = l[this' \leftarrow nil]) \Rightarrow \\
&\quad (\exists S',f' \bullet (this' \notin S \cup \{nil\}) \wedge (S' = S \cup \{this'\}) \wedge (f' = f[this' \leftarrow false]) \wedge \\
&\quad (S' \subseteq LS') \wedge (\forall s \in S' \bullet f'(s) = (l'(s) \neq nil))) \\
= &(S \subseteq LS) \wedge (\forall s \in S \bullet f(s) = (l(s) \neq nil)) \wedge (this' \notin LS \cup \{nil\}) \wedge \Rightarrow \\
&\quad (this' \notin S \cup \{nil\}) \wedge (S \cup \{this'\} \subseteq LS \cup \{this'\}) \wedge \\
&\quad (\forall s \in S \cup \{this'\} \bullet f[this' \leftarrow false](s) = (l[this' \leftarrow nil](s) \neq nil)) \\
= &\ true
\end{aligned}$$

For procedure *full* we immediately apply (8):

$$\begin{aligned}
&S.full \sqsubseteq_R LS.full \\
= &((this \in S) \wedge (S \subseteq LS) \wedge (\forall s \in S \bullet f(s) = (l(s) \neq nil)) \Rightarrow (this \in LS)) \wedge \\
&((this \in S) \wedge (S \subseteq LS) \wedge (\forall s \in S \bullet f(s) = (l(s) \neq nil)) \Rightarrow \\
&\quad (f(this) = (l(this) \neq nil))) \\
= &\ true
\end{aligned}$$

We rewrite procedure procedure *S.store* using the definitions. For procedure *LS.store* we expand the call, rename the local variable to *t*, apply (1) and (2) to merge the assignments, and apply (3) to eliminate the local variable:

$$\begin{aligned}
S.store &= \{this \in S\}\ ;\ this,S,f := this,S,f[this \leftarrow false])\} \\
LS.store &= \{this \in LS\}\ ;\ this,LS,l := \{this',LS',l' \mid (this' = this) \wedge \\
&\qquad (\exists t \bullet (t \notin LS \cup \{nil\}) \wedge (LS' = LS \cup \{t\}) \wedge \\
&\qquad (l' = l[t \leftarrow nil][this \leftarrow t]))
\end{aligned}$$

Refinement of *store* is established by applying (10); we leave out the details of the proof. To show condition (b.2) we first observe that $grd\ LS.new = true$, $grd\ LS.full = true$, and $grd\ LS.store = true$, i.e. all procedures are always enabled. Therefore condition (b.2) is immediately satisfied for all procedures. This completes the proof.

5 Concurrent Objects

Classes with actions are translated to modules with actions, such that there is one action for each object of the class. This is formally expressed by nondeterministically assigning any element of *C* to *this* before executing the action body. If *C* is empty, no action is enabled. For the time being we make the restriction that method calls can appear only as the first statement in methods and actions.

```
class $C$                                    $\widehat{=}$  module $C$
  attr $p : P$                                     var $C$ : set of $Object := \{\}$
  initialization $(g : G)$                         var $p : Object \to P$
    $I$                                            procedure $new(g : G,$ res $this : Object)$
  method $l(s : S,$ res $t : T)$                     $this :\notin C \cup \{nil\}$ ; $C := C \cup \{this\}$ ; $I$
    $L$                                            procedure $l(this : Object, s : S,$ res $t : T)$
  method $m(u : U,$ res $v : V)$                     $\{this \in C\}$ ; $L$
    $M$                                            procedure $m(this : Object, u : U,$ res $v : V)$
  action $a$                                         $\{this \in C\}$ ; $M$
    $A$                                            action $a$
  action $b$                                         var $this :\in C \bullet A$
    $B$                                            action $b$
end                                                  var $this :\in C \bullet B$
                                                 end
```

Inheritance and subtyping works as for classes without actions. Refinement of classes with actions translates to refinement of modules with actions. We give an example that illustrates the concept of delaying a computation by enabling a background action. Class *Doubler* allows to store an integer and to retrieve its double. Class *DelayedDoubler* doubles the integer in the background and blocks if the integer to be retrieved is not yet doubled:

```
class $Doubler$                              class $DelayedDoubler$
  attr $x : integer$                           attr $y : integer$
  method $store(u : integer)$                  attr $d : boolean$
    $this.x := 2 \times u$                     initialization $d := true$
  method $retrieve($ res $u : integer)$        method $store(u : integer)$
    $u := this.x$                                $y, d := u, false$
end                                            method $retrieve($ res $u : integer)$
                                                 when $d$ do $u := y$
                                               action $double$
                                                 when $\neg d$ do $y, d := 2 \times y, true$
                                             end
```

These classes translate to modules in the same fashion as previous examples. We give immediately the refinement relation needed to prove that *Doubler* is refined by *DelayedDoubler*:

$$\begin{array}{l} R(Doubler, x)(DelayedDoubler, y, d) = \\ \quad (Doubler = DelayedDoubler) \land \\ \quad (\forall o \in Doubler \bullet (d(o) \land y(o) = x(o)) \lor (\neg d(o) \land (2 \times y(o) = x(o))) \end{array}$$

We conclude this example by noting that we can alternatively express *DelayedDoubler* as a subtype of *Doubler*, thus arriving at a class hierarchy in which concurrency is introduced in a subclass:

```
**class** $DelayedDoubler$ **inherit** $Doubler$
  **attr** $d : boolean$
  **initialization** $d := true$
  **method** $store(u : integer)$
    $x, d := u, false$
  **method** $retrieve($ **res** $u : integer)$
    **when** $d$ **do** $u := x$
  **action** $double$
    **when** $\neg d$ **do** $x, d := 2 \times x, true$
**end**
```

Statements in classes are atomic only up to method calls. If method calls appear not only as the first statement in methods and actions, the class has to be *normalized* first. A method or action body with such a call has to be split in order to model that execution can block at that point. If at the point of the method call there are no local variables, then we introduce an auxiliary integer variable that is initialized to zero and incremented at the point of the method call. For every call we also introduce an action that contains the call and the remainder of the body. This action is enabled if the counter for that call is positive and the action decrements the counter first. We illustrate this by an example of a faulty merger:

```
**class** $FaultyMerger$
  **attr** $in1, in2, out : Buffer$
  **attr** $x1, x2 : integer$
  **initialization** $(i1, i2, o : Buffer)$
    $in1, in2, out := i1, i2, o$
  **action begin** $in1.get(x1)$ ; $out.put(x1)$ **end**
  **action begin** $in2.get(x2)$ ; $out.put(x2)$ **end**
**end**
```

Class *FaultyMerger* is normalized as follows:

```
**class** $FaultyMerger$
  **attr** $in1, in2, out : Buffer$
  **attr** $x1, x2 : integer$
  **attr** $at1, at2 : integer$
  **initialization** $(i1, i2, o : Buffer)$
    $in1, in2, out := i1, i2, o$
  **action begin** $in1.get(x1)$ ; $at1 := at1 + 1$ **end**
  **action when** $at1 > 0$ **do begin** $at1 := at1 - 1$ ; $out.put(x1)$ **end**
  **action begin** $in2.get(x2)$ ; $at2 := at2 + 1$ **end**
  **action when** $at2 > 0$ **do begin** $at2 := at2 - 1$ ; $out.put(x2)$ **end**
**end**
```

If $in1$ contains sufficiently many elements, the action $in1.get(x1)$; $at1 := at1 + 1$ can be taken several times and overwriting $x1$ before the action for placing x in *out* is taken. The class *Merger* avoids this problem with the help of an extra variable.

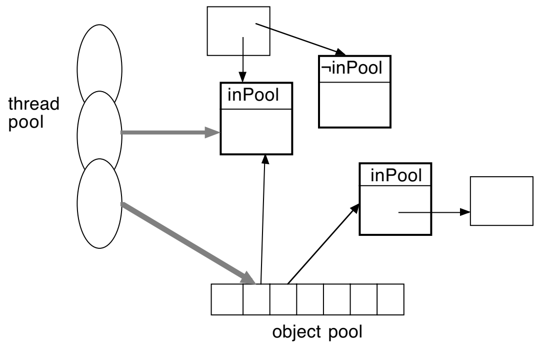

Fig. 1. Illustration of the implementation. Boxes with the *inPool* attribute represent active objects, the other passive objects. A thin arrow between boxes represents a reference, a thick arrows from a thread to an object represents a reference with a lock.

Suppose there are local variables at the point of the method call. These local variables form the context in which execution may resume, after possible interleaving with other methods or action. This is modelled by storing the context in an attribute with each object. As multiple activities may create local contexts, but the order of creation is ignored, the contexts are stored in a bag. We illustrate this by normalizing the action *notifyOneObserver* of class *Subject*:

```
attr $at1$ : bag of $Observer$
action $notifyOneObserver$
  when $notifyObs \neq \{\}$ do
    var $o : Observers$ •
    begin $o :\in notifyObs$ ; $notifyObs := notifyObs - \{o\}$ ; $at1 := at1 + [o]$ end
action $notifyOneObserver$
  var $o :\in at1$ • begin $at1 := at1 - [o]$ ; $o.update$ end
```

This normalization step is required before verification and refinement can be carried out by translating classes to modules.

6 Implementation

In order to test our ideas, we have developed a prototypical compiler for our language, see [17] for details. The compiler currently translates to the Java Virtual Machine. We sketch the principles of the implementation, see Fig. 1 for an illustration. The implementation relies on the restriction that method and action guards may refer only to attributes of the object itself and may not contain method calls. An object that has guarded methods is called a *guarded object*. An object that has actions is called an *active object*, otherwise a *passive object*. An active object that has at least one enabled action is called an *enabled object*, otherwise a *disabled object*.

At runtime a *thread pool* and an *object pool* are maintained. The object pool is initially empty. When an active object is created, a pointer to it is placed in the object pool and only active objects are placed in the object pool. Each active object has an extra boolean attribute *inPool* indicating whether a pointer to it is in the object pool. Threads request a reference to an active object from object pool. If the object is disabled, the thread resets the *inPool* attribute and removes it from the object pool. If the object is enabled, the thread executes an enabled action and leaves the object in the object pool. Each thread obtains a lock to an object when entering one of its methods or actions and releases the lock when exiting the method or action. The lock is also released at a call to another object and obtained again at re-entry from the call. If a guarded method is called the guard is evaluated and the thread waits if the guard is false. At the exit from a guarded object all waiting threads are notified to reevaluate the guards.

Fairness among the actions of an object is ensured by evaluating the guards in a cyclic fashion. This is done with one additional attribute for the index of the last evaluated action guard in every active object. The object pool is implemented as a dynamic array. Fairness among the objects is ensured by retrieving active objects in a cycling fashion. The object pool grows and shrinks like a stack: new objects are added at the end and when an object is retrieved, its position is filled with the last object. Hence adding objects and retrieving objects take constant time. Active objects are garbage collected like passive objects, i.e. when there is no reference from any other object and no reference from the object pool.

With this scheme action guards are only evaluated when a thread is searching for an action to execute. Method guards are only re-evaluated when another thread has exited the object and thus possibly affected the guard. The memory overhead is that every active object requires one bit for the *inPool* attribute, one integer for the index to the last evaluated action guard, and one pointer in the object pool. We are currently experimenting with techniques to control the creation and termination of threads.

7 Discussion

A number of attempts have been made in formalizing objects with records, initiated by the work of Cardelli [12] and leading to various type systems incorporating object-oriented concepts. Our experience in using one such type system is that in verification and refinement it is more in the way than helpful [22]. Understanding attributes as mappings from object identities to their values emerges naturally in object modeling techniques like [21], and is used in a number of formalizations of object models, e.g. [15]. The approach of viewing a method as a procedure with an additional *this* parameter is also taken in Modula-3 and Oberon-2. We find that this combination leads to a simple model with a clear distinction between classes and types. A consequence is that objects can only be allocated on the heap, an approach also taken in several mainstream object-oriented languages.

One may argue about releasing the lock to an object when a method call in that object goes to another object, hence allowing other methods to be called or actions to be initiated. Indeed in our first implementation we retained the lock. However we found programs to be difficult to analyze, as it is necessary to keep track of which objects are locked by which actions. The model of releasing the lock allows some disciplined intra-object concurrency: while several actions or methods can be initiated, only one can progress, thus still guaranteeing atomicity of attribute updates. In order for a class invariant to be preserved, the class invariant has not only to be established at the end of every method, but also before each call to another object. The need for doing so is already recognized in sequential programs when re-entrance is possible [19].

The model presented does not define (indirectly) recursive method calls; doing so would require taking a fixed point. The model also does not accurately capture how self- and super-calls are resolved when methods are redefined: a super-call will always remain in the superclass, even if calling other methods that are redefined in the subclass. In order to model this, methods calls must not be resolved immediately, but when objects of the classes are created. A model of inheritance that delays resolution of method calls to the time when objects are created was proposed by Cook and Palsberg [13] and applied to studies of class refinement by Mikhajlov and Sekerinski [18]. While our implementation follows this model, the presented theory does not capture this.

Acknowledgement

The author's understanding profited from the interaction with participants of FMCO 02; discussions with Rustan Leino are particularly acknowledged. The comments of the reviewer lead to a significant improvement.

References

1. Pierre America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.
2. Pierre America and Frank de Boer. Reasoning about dynamically evolving process structures. *Formal Aspects of Computing*, 6(3):269–316, 1994.
3. Ralph Back. Refinement calculus, part II: Parallel and reactive programs. In J. W. deBakker, W.-P. deRoever, and G. Rozenberg, editors, *REX Workshop on Stepwise Refinement of Distributed Systems - Models, Formalisms, Correctness*, Lecture Notes in Computer Science 430, pages 67–93, Mook, The Netherlands, 1989. Springer Verlag.
4. Ralph Back, Martin Büchi, and Emil Sekerinski. Action-based concurrency and synchronization for objects. In T. Rus and M. Bertran, editors, *Transformation-Based Reactive System Development, Fourth AMAST Workshop on Real-Time Systems, Concurrent, and Distributed Software*, Lecture Notes in Computer Science 1231, pages 248–262, Palma, Mallorca, Spain, 1997. Springer-Verlag.
5. Ralph Back and Kaisa Sere. Action systems with synchronous communication. In E.-R. Olderog, editor, *IFIP Working Conference on Programming Concepts, Methods, Calculi*, pages 107–126, San Miniato, Italy, 1994. North-Holland.
6. Ralph Back and Joakim von Wright. Trace refinement of action systems. In B. Jonsson and J. Parrow, editors, *CONCUR '94: Concurrency Theory*, Lecture Notes in Computer Science 836. Springer-Verlag, 1994.
7. Ralph Back and Joakim von Wright. *Refinement Calculus – A Systematic Introduction*. Springer-Verlag, 1998.
8. Marcello M. Bonsangue, Joost N. Kok, and Kaisa Sere. An approach to object-orientation in action systems. In *Mathematics of Program Construction*, Lecture Notes in Computer Science 1422, Marstrand, Sweden, 1998. Springer-Verlag.

9. Marcello M. Bonsangue, Joost N. Kok, and Kaisa Sere. Developing object-based distributed systems. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *3rd IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'99)*, pages 19–34. Kluwer, 1999.
10. Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, 1998.
11. Martin Büchi and Emil Sekerinski. A foundation for refining concurrent objects. *Fundamenta Informaticae*, 44(1):25–61, 2000.
12. Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *International Symposium on the Semantics of Data Types*, Lecture Notes in Computer Science 173, pages 51–67. Springer-Verlag, 1984.
13. William Cook and Jens Palsberg. A denotational semantics of inheritence and its correctness. In *ACM Conference Object Oriented Programming Systems, Languages and Applications*, ACM SIGPLAN Notices, Vol 14, No 10, pages 433–443, 1989.
14. Steve J. Hodges and Cliff B. Jones. Non-interference properties of a concurrent object-based language: Proofs based on an operational semantics. In Burkhard Freitag, Cliff B. Jones, Christian Lengauer, and Hans-Joerg Schek, editors, *Object Orientation with Parallelism and Persistence*, pages 1–22. Kluwer Academic Publishers, 1996.
15. Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.
16. Cliff B. Jones. Accomodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.
17. Kevin Lou. *A Compiler for an Action-Based Object-Oriented Programming Language*. Master's thesis, McMaster University, 2003.
18. Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In Eric Jul, editor, *ECOOP'98 – 12th European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science 1445, pages 355–382, Brussels, Belgium, 1998. Springer-Verlag.
19. Leonid Mikhajlov, Emil Sekerinski, and Linas Laibinis. Developing components in presence of re-entrance. In J. Wing, J. Woodcock, and J. Davis, editors, *World Congress on Formal Methods, FM'99*, Lecture Notes in Computer Science 1709, Toulouse, France, 1999. Springer-Verlag.
20. Jayadev Misra. A simple, object-based view of multiprogramming. *Formal Methods in System Design*, 20(1):23–45, 2002.
21. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddi, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.
22. Emil Sekerinski. A type-theoretic basis for an object-oriented refinement calculus. In S. J. Goldsack and S. J. H. Kent, editors, *Formal Methods and Object Technology*, pages 317–335. Springer-Verlag, 1996.
23. Kaisa Sere and Marina Waldén. Data refinement of remote procedures. *Formal Aspects of Computing*, 12(4):278–297, 2000.