Verification and Refinement with Fine-Grained Action-Based Concurrent Objects

Emil Sekerinski

McMaster University, Hamilton, Ontario
emil@mcmaster.ca

Abstract

Action-based concurrent object-oriented programs express autonomous behavior of objects through actions that, like methods, are attached to objects but, in contrast to methods, may execute autonomously whenever their guard is true. The promise is a streamlining of the program structure by eliminating the distinction between processes and objects and a streamlining of correctness arguments. In this paper we illustrate the use of action-based object-oriented programs and study their verification and their refinement from specifications, including the issue of non-atomic operations.

Key words: object-based concurrency, actions, verification, refinement, weakest preconditions

1 Introduction

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: concurrency is expressed in terms of processes (threads) that have to be managed separately from objects.

Action-based object-oriented concurrency offers the promise of truly integrating objects and concurrency by eliminating the need for having the class structure and the process structure as two interdependent design views. The only syntactic additions needed are extending classes by actions, which execute autonomously, and allowing methods to be guarded. In this way, concurrency can be introduced in subclasses of a class hierarchy by adding (new) actions.

For example, this permits concurrency to be treated as an implementation issues that can be delegated to subclasses.

The approach to action-based object-oriented concurrency taken here is that (1) atomicity of operations (methods and actions) is guaranteed only up to method calls (2) several operations can be initiated in one object, but only one can progress at any time, and (3) actions, like methods, can be initiated multiple times. In combination, this leads to a fine-grained model of concurrency allowing a higher degree of concurrency than the exclusive access to an object for the entire duration of an operation, as is sometimes associated with monitors and concurrent objects.

An earlier paper [17] gives the formal model of the language in terms of higher order predicate transformers and also sketches the current implementation. The compiler translates to the Java Virtual Machine and is described in more detail in [15].

The purpose of this paper is to further streamline the formal verification and refinement process: all reasoning is reduced to Dijkstra's (syntactic) weakest precondition predicate transformer through (syntactic) transformations. Most of these transformations are meant to be simple enough that they can be done on the fly. In essence, an attribute of a class is understood as function mapping objects of that class to attribute values, methods are understood as procedures taking an additional *this* parameter, and actions are quantified over all objects of the class. Difficulties arise with non-atomic operations: each atomic region has to be transformed into a separate action and local variables that are present in the context need to be stored in bag-valued attributes. While not all aspects of the language can be dealt with in this way, this paper demonstrates through a series of examples what can be achieved. Inheritance, subtyping, type tests, type casts, and dynamic binding can be dealt with as in [17]; here we focus on concurrency.

Briot et. al. [9] give a classification of concurrency in object-oriented programming, based on the *level of concurrency*, *autonomy of objects*, and the *acceptance of messages*. The level of concurrency can be classified here as *quasi-concurrent*, like in ABCL/1 [19], as several method activations may co-exist, but at most one is not suspended; it is a disciplined form of *intra-object concurrency*. This is in contrast to *serial* objects like in POOL [2, 3] that support only one method activation and fully concurrent objects like with Actors [1]. Our objects would be classified as *autonomous* rather than *reactive* as they may be active without receiving a method call; in Java all objects are reactive and autonomous activity is expressed through threads. The acceptance of messages is *implicit* rather than *explicit* as in Ada and POOL; in those languages each object has a *body* that controls entry into the object through a *rendezvous*. Here, condition synchronization is achieved through *guards* instead. The communication between objects is through *synchronous method calls*, as in Ada, POOL, and Java, rather than through *message queues* as in Actors.

The closest work is the Seuss approach of Misra [16] and OO-action systems of Bonsangue, Kok and Sere [7, 8]. We share with these approaches the use of synchronous method calls, the use of guards for condition synchronization, and the use of actions to express autonomous activity. While in Seuss only a fixed number of objects can be declared, we allow dynamic object creation, as in OO-action systems. A notable difference is how atomicity of actions and methods is guaranteed if they contain multiple method calls. Suppose we have an (unguarded) action $x.m\ ;\ y.n$ and method n of object y is not enabled. In OO-action systems, following the theory of action systems [6, 18], the whole action is therefore not enabled; thus, if we would have executed $x.m$, we would have to roll back. In Seuss this is solved by allowing one call to a guarded method and that has to be the first statement in an action or a method. Besides being a syntactic restriction, this forbids that an unguarded method is refined by a guarded, as done in OO-action systems. We do not have this restriction, but allow that an action or method gets suspended at the point where a method is called. That is, actions and methods are atomic only up to method calls.

Much of the inspiration comes from the $\pi o \beta \lambda$ approach that was initiated by Jones [13, 14], even though $\pi o \beta \lambda$ is defined in terms of the π calculus and our language is defined in terms of action systems. We do not directly support *early return* and *delegate* statements as $\pi o \beta \lambda$ does, though an early return can be expressed through actions and guarded methods. Hoare-style verification rules for a POOL-like language that includes statements for sending and receiving synchronous messages are given in [4].

The next section introduces the language through a series of examples, starting with a definition of the formal syntax. Section 3 presents the verification and Section 4 the refinement of classes with atomic operations. Section 5 extends the treatment to non-atomic actions. We conclude with a discussion of implementation aspects and critical remarks in Section 6.

2 An Action-Based Concurrent Object-Oriented Language

We start by giving the 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:

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

A class is declared by giving it a name and then listing all the attributes (instance variables), initializations, methods, and actions. Initializations have only value parameters, methods may have both value parameters and return a result, and actions don't have parameters. While the syntax allows multiple initializations, we only consider classes with at most one declared initialization. Methods of a class may have the same name, as long as the methods differ in the type of their parameters. Actions are named and the names must be unique, though the name does not carry any meaning. Both methods and actions may optionally have a *guard*, a Boolean expression that must be only over attributes of the object itself. A method or action is *enabled* if its guard is true or missing, otherwise it is *disabled*. An object is *active* if its class defines some actions; otherwise it is *passive*. The assertion statement **assert** b does nothing if b hold and aborts if b does not hold. The assignment $x := e$ assigns simultaneously the values of the list e to the list x of variables. The nondeterministic assignment statement $x :\in s$ generalizes this to selecting values of (the tuple) x such that $x \in s$. This statement is not part of the programming language, but is included here for use in abstract programs. A method call $x := c.m(e)$ to object c takes the list e as the value parameters and assigns the result to x. The object creation $c :=$ **new** $C(e)$ creates a new object of class C and calls the corresponding initialization with value parameters e. We do not elaborate the structure of *identifier*, *expression*, *type*, and *designator*.

We introduce the language through a series of examples, starting with an example for active objects: an aquarium in which fish move randomly to the left and to the right. The main program creates seven fish objects; once a fish object is created, any of its enabled actions can be selected for execution. In case more than one is enabled, the choice is nondeterministic (in case no action is enabled and no reference to an object exists, the object can be garbage collected). As the bodies of all actions (and methods) access only attributes of the object itself, actions of any two fish objects can be executed in parallel, though only one action or method of a fish can be executed at any time:

class *Fish*
  var x, d : *integer*
  var r : *boolean*
  initialization $x, d, r := 0, 5, \mathit{true}$
  method *setPace*(p : *integer*)
    begin assert $p > 0$; $d := p$ **end**
  action *moveRight*
    when $x + d < W \wedge r$ **do** $x := x + d$
  action *moveLeft*
    when $x - d \geq 0 \wedge \neg r$ **do** $x := x - d$
  action *changeToRight*
    when $x < W - 1 \wedge \neg r$ **do** $r := \mathit{true}$
  action *changeToLeft*
    when $x > 0 \wedge r$ **do** $r := \mathit{false}$
end

var f : *Fish*, n : *integer* ;
begin $n := 0$; **while** $n < 7$ **do** $f :=$ **new** *Fish* **end**

The next example illustrates the use of guarded methods: in a class for a bounded buffer, the guards protect the buffer from overflow and underflow. Calling a method that is disabled *blocks* execution at that point until the guard becomes true, as in the actions of class *Merger* below. As all objects are of type *Object*, any object can be placed in the buffer:

class *Buffer*
  var b : **array** M **of** *Object*
  var $\mathit{in}, \mathit{out}, n$: *integer*
  initialization
    $\mathit{in}, \mathit{out}, n := 0, 0, 0$
  method *put*(x : *Object*)
    when $n < M$ **do** $\mathit{in}, b[\mathit{in}], n := (\mathit{in} + 1)$ **mod** $\mathit{max}, x, n + 1$
  method *get* : *Object*
    when $n > 0$ **do** $\mathit{out}, \mathit{result}, n := (\mathit{out} + 1)$ **mod** $\mathit{max}, b[\mathit{out}], n - 1$
end

```
class Merger
  var in1, in2, out : Buffer
  var a1, a2 : boolean
  var x1, x2 : Object
  initialization (i1, i2, o : Buffer)
    in1, in2, out, a1, a2 := i1, i2, o, true, true
  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, actions *copy*1 and *copy*2 are both enabled. If *copy*1 is invoked, the execution may block at the call $in1.get(x1)$ or at the call $out.put(x1)$. In general, method and action bodies are atomic only up method calls: the guard is evaluated and all statements up to the first method call are executed atomically; all subsequent statements up to the next method call are also executed atomically. Arbitrary many activities, i.e. method calls or action invocations can be initiated in one object, including multiple initiations of the same method or action, but only one can progress at any time. Here, both *copy*1 and *copy*2 can be initiated. As both actions disable themselves after initiation and remain disabled until completion, they cannot be initiated a second time.

The next example shows the use of *semaphores* for achieving fairness. In general, the choice of guards for evaluation is not bound to a fairness policy.

```
class Semaphore
  var n : integer
  initialization (c : integer)
    n := c
  method P
    when n > 0 do n := n − 1
  method V
    n := n + 1
end
```

```
class StrongSemaphore
  var n : integer
  var q : seq of Object
  initialization (c : integer)
    n, q := c, ⟨⟩
  method P(u : Object)
    begin q := q ∘ ⟨u⟩ ; Q(u) end
  method Q(u : Object)
    when n > 0 ∧ u = head(q) do
      n, q := n − 1, tail(q)
  method V
    n := n + 1
end
```

Considering an object s of class *Semaphore*, a sequence $s.P$; ... *critical section* ... ; $s.V$ in object x might never enter the critical section while the same sequence from another object may continuously do so. A *strong semaphore* ensures a first-in first-out policy by keeping a sequence of requests. For *ss* of class *StrongSemaphore*, a typical use would be $ss.P(this)$; … *critical section* …; $ss.V$, where *this* is the reference to the current object.

The example of the dining philosophers is well-known. We represent philosophers by active objects and forks by passive objects; philosophers have two actions, one for the transition from *thinking* to *eating* and one for the transition from *eating* to *thinking*; forks become binary semaphores. The main program connects the philosophers and forks in a cyclic fashion. As known, in this way the situation may occur that all philosophers pick up their left fork and no philosopher gets a chance to eat.

```
class Phil
  var state : (thinking, hungry, eating, full)
  var left, right : Fork
  initialization (l, r : Fork)
    state, left, right := thinking, l, r
  action needToEat
    when state = thinking do
      begin state := hungry ;
        left.pickUp ; right.pickUp ;
        state := eating
      end
  action needToThink
    when state = eating do
      begin state := full ;
        left.putDown ; right.putDown ;
        state := thinking
      end
end
```

```
class Fork
  var available : boolean
  initialization
    available := true
  method pickUp
    when available do
      available := false
  method putDown
    available := true
end
```

```
var fork : array 5 of Fork ;
var phil : array 5 of Phil ;
var i, j : integer ;
begin i, j := 0, 0 ;
  while i < 5 do fork[i] := new Fork ;
  while j < 5 do phil[j] := new Phil(fork[j], fork[(j + 1) mod 5])
end
```

A *priority queue* offers a method $add(e)$ for storing integer e, a method *remove* for removing the least integer stored so far, and a method *empty* for testing whether the priority queue is empty. Our implementation is by a linked list of nodes. Elements are stored in attribute m in ascending order (duplicates are allowed). Attribute l points to the next node or is *nil* at the last object, which does not hold a queue element. An element is added to the priority queue by either storing it in the current node if it is the last one (and creating a new last node), or by depositing it in the current node and enabling an action that will move either the new element or the element of the current node one position down. The least element is removed by returning the element of the current node immediately and enabling an action that will move the element of the next node one position up, or set the l pointer to *nil* if the node becomes the last one. The Boolean attributes i, a, r reflect whether the queue element is idle, an addition is requested, and a removal is requested, respectively:

```
class PriorityQueue
  var m, p : integer
  var l : PriorityQueue
  var i, a, r : boolean
  initialization l, i, a, r := nil, true, false, false
  method empty : boolean
    result := l = nil
  method add(e : integer)
    when i do
      if l = nil then
        begin m := e ; l := new PriorityQueue end
      else
        p, i, a := e, false, true end
  method remove : integer
    when i do
      result, i, r := m, false, true
  action doAdd
    when a do
      begin a := false ;
        if m > p then m, p := p, m ;
        l.add(p) ;
        i := true
      end
  action doRemove
    when r do
      begin r := false
        if l.empty then l := nil
        else m := l.remove ;
        i := true
      end
end
```

The guards of the *PriorityQueue* methods and actions are such that at most one method or action can execute at any time. Thus a priority queue can have at most as many concurrent actions as there are nodes in the queue. The concurrent behavior of *PriorityQueue* is such that it is not observable through calls to the methods *empty*, *add*, and *remove*. It is an example where concurrency is introduced for efficiency. Formally, we claim that *PriorityQueue* is a refinement of class *PriorityBag*, which uses a bag (multiset) to abstractly represent it's state. Let [] stand for the empty bag, $[e]$ for the bag containing only e, binary operator $+$ for bag addition, $-$ for bag subtraction, and $min(b)$ for the least element of bag b:

> **class** *PriorityBag*
>   **var** b : **bag of** *integer*
>   **initialization** $b := []$
>   **method** *empty* : *boolean*
> $result := b = []$
>   **method** $add(e : integer)$
> $b := b + [e]$
>   **method** *remove* : *integer*
> $b, result := b - [min(b)], min(b)$
> **end**

The final 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*
>   **var** *sub* : *Subject*
>   **initialization** $(s : Subject)$
>   **begin** $sub := s$; $s.attach(this)$ **end**
>   **method** *update* ...
> **end**
>
> **class** *Subject*
>   **var** a, n : **set of** *Observer*
>   **initialization** $a, n := \{\}, \{\}$
>   **method** $attach(o : Observer)$
> $a := a \cup \{o\}$
>   **method** *notifyAll*
> $n := a$
>   **action** *notifyOne*
>   **when** $n \neq \{\}$ **do**
>   **var** o : *Observer* ;
>   **begin** $o :\in n$; $n := n - \{o\}$; $o.update$ **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 *notifyAll*.

We conclude the introduction of the language with a comparison of objects and monitors [5, 11]. Both objects and monitors guarantee exclusive access to private data, though compared to traditional monitors (and to Java), there are no condition variables, no *signal* and *wait* operations, and no processes (or threads) as explicit language constructs—their role is taken over by guarded methods and actions. Method calls from one object to other objects—the equivalent of nested monitor calls—are *open* as the exclusive access to the first object is dropped and only regained when the call returns. By comparison, method calls in Java (with appropriate synchronization) are *closed* as exclusive access to all objects in the call chain is retained. It is known that closed calls allow less concurrency and are more prone to deadlocks. On the other hand, open calls require the class invariant to be established at each call that leaves an object. This may include disabling those methods and actions that would otherwise not preserve the invariant, as for example in class *PriorityQueue*.

3 Verification

For analyzing the correctness of programs we consider a simpler *kernel language of atomic statements*. All we need to assume is that all atomic statements are characterized by Dijkstra's weakest precondition predicate transformer: $wp(S, c)$ is the weakest precondition such that S terminates and establishes postcondition c. Moreover, we assume that all statements are monotonic, i.e. for any statement S and any Boolean expressions b, c:

$$(b \Rightarrow c) \Rightarrow (wp(S, b) \Rightarrow wp(S, c)) \tag{1}$$

We define some basic statements: the *assertion statement* $\{b\}$, the *assumption* or *guard statement* $[b]$, the multiple assignment $x := e$, the nondeterministic assignment $x :\in s$, the nondeterministic choice $S \sqcap T$ between statements S and T, and the unbounded choice $\sqcap x \in s \bullet S$. Further statements, like iteration, can be added. All variables are assumed to have a unique type, even though it is commonly omitted. With x a list of variables and e a list of expressions, we write $f[x \backslash e]$ for expression f with all free occurrences of x substituted by e. For Boolean expressions, $\equiv$ has the same meaning as $=$, though $\equiv$ binds weaker than all other Boolean operators. For the time being, we assume that the evaluation of all expressions succeeds:

$$wp(\{b\}, c) \quad \equiv b \wedge c \tag{2}$$
$$wp([b], c) \quad \equiv b \Rightarrow c \tag{3}$$

$$
\begin{aligned}
&wp(x := e, c) && \equiv c[x \backslash e] && && (4)\\
&wp(S\ ;\ T, c) && \equiv wp(S, wp(T, c)) && && (5)\\
&wp(S \sqcap T, c) && \equiv wp(S, c) \wedge wp(T, c) && && (6)\\
&wp(\sqcap x \in s \bullet S, c) && \equiv (\forall x \in s \bullet wp(S, c)) && x \text{ not free in } c && (7)
\end{aligned}
$$

The nondeterministically initialized local variable declaration **var** $x \in s\ ;\ S$ stands for $\sqcap x \in s \bullet S$. The local variable declaration **var** $x : T\ ;\ S$ stands for $\sqcap x \bullet S$, where x ranges over all elements of type T. The nondeterministic assignment $x :\in s$ stands for $\sqcap h \in s \bullet x := h$. 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. The assertion statement **assert** b is synonymous to $\{b\}$. The guarded statement **when** b **do** S and the conditional statements are defined as:

$$
\begin{aligned}
&\textbf{when } b \textbf{ do } S && \mathrel{\widehat{=}} [b]\ ;\ S && (8)\\
&\textbf{if } b \textbf{ then } S && \mathrel{\widehat{=}} ([b]\ ;\ S) \sqcap [\neg b] && (9)\\
&\textbf{if } b \textbf{ then } S \textbf{ else } T && \mathrel{\widehat{=}} ([b]\ ;\ S) \sqcap ([\neg b]\ ;\ T) && (10)
\end{aligned}
$$

As derived rules we get:

$$
\begin{aligned}
&wp(x :\in s, c) && \equiv (\forall x \in s \bullet c) && (11)\\
&wp(\textbf{when } b \textbf{ do } S, c) && \equiv b \Rightarrow wp(S, c) && (12)
\end{aligned}
$$

In programs, evaluation of expressions may fail. While in the logic any expression always has a value of its type, undefinedess of expressions in statements needs to be taken into account. For a program expression e, let $\Delta\, e$ stand for the definedness of e. For example, we have that $\Delta(x \textbf{ div } y) \equiv y \neq 0$. That is, Δ can be defined over the syntactic structure of program expressions. For a statement to terminate evaluation of all expressions must succeed; we define the weakest preconditions for statements with possibly undefined expressions accordingly:

$$
\begin{aligned}
&wp(\{b\}, c) && \equiv \Delta\, b \wedge b \wedge c && (13)\\
&wp([b], c) && \equiv \Delta\, b \wedge (b \Rightarrow c) && (14)\\
&wp(x := e, c) && \equiv \Delta\, e \wedge c[x \backslash e] && (15)
\end{aligned}
$$

The declaration of a class C amounts to the declaration of a global variable C for the set of all objects of class C and for each attribute f of type F, a global variable $C.f$ mapping objects of C to values of F:

$$
\begin{aligned}
&\textbf{var } C : \textbf{set of } Object && (16)\\
&\textbf{var } C.f : Object \to F && (17)
\end{aligned}
$$

That is, we use the class name also for the set of objects of that class and as a prefix of the attribute names. We assume that the type *Object* contains infinitely many elements, including the distinguished element *nil*. The notation **set of** T stands for finite sets of type T. We commonly drop the prefix and write f for $C.f$, if there is no ambiguity. Accessing an attribute f of object o, written $o.f$ amounts to applying the function f to o. An attribute assignment amounts to a function update:

$$
\begin{aligned}
o.f &= f(o) && (18)\\
o.f := e &= f := f[o \leftarrow e]) && (19)
\end{aligned}
$$

We write $f[a \leftarrow r]$ for modifying function f to return r for argument a, formally:

$$
\begin{aligned}
&a.f[a \leftarrow r] = r && (20)\\
&b.f[a \leftarrow r] = b.f, \quad b \neq a && (21)
\end{aligned}
$$

The nondeterministic assignment $x := ?$ assigns to x an arbitrary value of its type. Defined as $\sqcap h \bullet x := h$, we have:

$$
\begin{aligned}
wp(x := ?, c) &\equiv (\forall x \bullet c) && (22)\\
wp(o.f := ?, c) &\equiv (\forall h \bullet c[f \backslash f[o \leftarrow h]]) && (23)
\end{aligned}
$$

The *enabledness domain* or *guard* of S is defined by $grd\ S = \neg wp(S, false)$ and the *termination domain* by $trm\ S = wp(S, true)$. For example, we have:

$$
\begin{aligned}
grd(\{b\}\ ;\ S) &\equiv grd\ S && (24)\\
grd([b]\ ;\ S) &\equiv b \wedge grd\ S && (25)\\
grd(\sqcap x \in s \bullet S) &\equiv (\exists x \in s \bullet grd\ S) && (26)\\
trm(\{b\}\ ;\ S) &\equiv b \wedge trm\ S && (27)\\
trm([b]\ ;\ S) &\equiv grd\ S && (28)
\end{aligned}
$$

Assume I is the body of the initialization of class C, or *skip* if no initialization is declared, M is the body of method *meth* of C, and A is the body of action *act*. We let $C.init$ stand for $this.a := ?\ ;\ I$, where a are the attributes that are not assigned to in I (a programming language may impose the syntactic restriction that all attributes have to be initialized, making this convention unnecessary). The declaration of class C induces following definitions, for each method *meth* and action *act*:

$$
\begin{aligned}
C.new &= this :\notin C \cup \{nil\}\ ;\ C := C \cup \{this\}\ ;\ C.init && (29)\\
C.meth &= \{this \in C\}\ ;\ M && (30)\\
C.act &= (\sqcap this \in C \bullet A) && (31)
\end{aligned}
$$

That is, we use the class name also as a prefix for the method and actions names. We let $x :\notin s$ stand for $x :\in \overline{s}$, where $\overline{s}$ is the complement of set s. The definition of $C.act$ in terms of a nondeterministic choice models concurrency through *interleaving*: if two actions operating on a disjoint state space are enabled, they can be executed in any order or in parallel.

For example, the declaration of class *Fish* gives rise to a global variable *Fish* with the identities of all *Fish* objects and variables $Fish.x$, $Fish.d$, $Fish.r$—

further on referred to by x, d, r—mapping each *Fish* object to the corresponding attribute values:

$$
\begin{array}{l}
\textbf{var } \mathit{Fish} : \textbf{set of } \mathit{Object} \\
\textbf{var } \mathit{Fish}.x, \mathit{Fish}.d : \mathit{Object} \to \mathit{integer} \\
\textbf{var } \mathit{Fish}.r : \mathit{Object} \to \mathit{boolean}
\end{array}
$$

We commonly abbreviate the reference *this.f* to an attribute of the current object by f. Making references to *this* explicit, we have for class *Fish*:

$$
\begin{array}{lcl}
\mathit{Fish.new} & = & \mathit{this} :\notin \mathit{Fish} \cup \{\mathit{nil}\} \ ; \ \mathit{Fish} := \mathit{Fish} \cup \{\mathit{this}\} \ ; \\
 & & \mathit{this}.x, \mathit{this}.d, \mathit{this}.r := 0, 5, \mathit{true} \\
\mathit{Fish.setPace} & = & \{\mathit{this} \in \mathit{Fish}\} \ ; \ \{p > 0\} \ ; \ \mathit{this}.d := p \\
\mathit{Fish.moveRight} & = & (\sqcap \mathit{this} \in \mathit{Fish} \bullet [\mathit{this}.x + \mathit{this}.d < W \wedge \mathit{this}.r] \ ; \\
 & & \quad \mathit{this}.x := \mathit{this}.x + \mathit{this}.d)
\end{array}
$$

Creating a new element of class C amounts to finding an unused element of C, adding that to C, and executing the body of the initialization. Assuming that v are the formal parameters of the initialization, we define:

$$
o := \textbf{new } C(e) = \textbf{var } \mathit{this}, v \ ; \ v := e \ ; \ C.\mathit{new} \ ; \ o := \mathit{this} \tag{32}
$$

In order to illustrate parameter passing with methods calls we define an *atomic* method call as follows. Suppose method m of class C is declared with value parameters v and to return a result. Then an *atomic* call $x := c.m(e)$ for $c \in C$ makes c and e to be the actual value parameters and x the actual result parameter:

$$
\begin{array}{l}
x := c.\mathit{meth}(e) = \textbf{var } \mathit{this}, v, \mathit{result} \ ; \\
\qquad \mathit{this}, v := c, e \ ; \ C.\mathit{meth} \ ; \ x := \mathit{result}
\end{array} \tag{33}
$$

Later on we consider non-atomic method calls, which require a prior transformation. Subtyping, inheritance, type test, and dynamic binding can be added as in [17]: if class D defines a subtype of C, then this amounts to stating that $D \subseteq C$ at any time. We do not go further into details as these constructs are not used later on.

While *wp* ensures total correctness, for invariance properties *partial correctness* is sufficient, motivating the introduction of *weakest liberal preconditions*. For a statement S, the predicate $\mathit{wp}(S, \mathit{true})$ is the weakest precondition for S to terminate, in whatever state. The weakest liberal precondition $\mathit{wlp}(S, c)$ is the weakest precondition for S to establish c provided S terminates, defined as $\mathit{wlp}(S, c) \equiv \mathit{trm}\ S \Rightarrow \mathit{wp}(S, c)$. In case all program expressions are defined we have:

$$
\mathit{wlp}(\{b\}, c) \equiv b \Rightarrow c \tag{34}
$$

$$
\mathit{wlp}([b], c) \equiv b \Rightarrow c \tag{35}
$$

$$
\mathit{wlp}(x := e, c) \equiv c[x \backslash e] \tag{36}
$$

$$wlp(S\ ;\ T, c) \Leftarrow wlp(S, wlp(T, c)) \tag{37}$$

In case program expressions are possibly undefined we have:

$$\begin{aligned} wlp(\{b\}, c) &\equiv \Delta\, b \wedge b \Rightarrow c && (38)\\ wlp([b], c) &\equiv \Delta\, b \wedge b \Rightarrow c && (39)\\ wlp(x := e, c) &\equiv \Delta\, e \Rightarrow c[x \backslash e] && (40) \end{aligned}$$

Definition 1 (Class Invariant) *Let C be a class in which the bodies of all initializations, methods, and actions are atomic, i.e. they do not contain (non-atomic) method calls. Boolean expression P is an invariant of C if following conditions hold:*

(a) Program Initialization: *When no objects exists, the invariant holds:*

$$C = \{\} \Rightarrow P$$

(b) Object Creation: *The object creation preserves the invariant:*

$$P \Rightarrow wlp(C.new, P)$$

(c) Methods: *Every method meth preserves the invariant:*

$$P \Rightarrow wlp(C.meth, P)$$

(d) Actions: *Every action act preserves the invariant:*

$$P \Rightarrow wlp(C.act, P)$$

These conditions are justified by appealing to the definition of classes in terms of actions systems with procedures [7, 10, 17]: if P is an invariant of a class, then P is also an invariant of the corresponding action system, and in any observable state, P will hold. As an example, we show that for class *Fish*, the predicate $0 \leq x < W$ is an invariant. In order to do so, we have to strengthen this expression to include $d > 0$ and have to quantify it over all objects of class *Fish*:

$$B \equiv (\forall f \in Fish \bullet f.d > 0 \wedge 0 \leq f.x < W)$$

Thus the conditions for B to be an invariant of *Fish* are:

(a) $Fish = \{\} \Rightarrow B$
(b) $B \Rightarrow wlp(Fish.new, B)$
(c) $B \Rightarrow wlp(Fish.setPace, B)$
(d.1) $B \Rightarrow wlp(Fish.moveRight, B)$
(d.2) $B \Rightarrow wlp(Fish.moveLeft, B)$
(d.3) $B \Rightarrow wlp(Fish.changeToRight, B)$
(d.4) $B \Rightarrow wlp(Fish.changeToLeft, B)$

Condition (a) amounts to a quantification over an empty range, which holds vacuously. For (b) we first expand *Fish.new* and B and then apply (37) and (36):

$$\begin{array}{ll} & wlp(Fish.new, B) \\ \Leftarrow & wlp(this :\notin Fish \cup \{nil\}\ ;\ Fish := Fish \cup \{this\}, \\ & \quad (\forall f \in Fish \bullet f.d[this \leftarrow 5] > 0 \wedge 0 \leq f.x[this \leftarrow 0] < W)) \\ \Leftarrow & this \notin Fish \cup \{nil\} \Rightarrow \\ & \quad (\forall f \in Fish \cup \{this\} \bullet f.d[this \leftarrow 5] > 0 \wedge 0 \leq f.x[this \leftarrow 0] < W) \\ \Leftarrow & (\forall f \in Fish \bullet f.d > 0 \wedge 0 \leq f.x < W) \\ \equiv & B \end{array}$$

The second last step follows from a case analysis with $this = f$ and $this \neq f$ and (20), (21). For (c) we proceed similarly, now applying also (34):

$$\begin{array}{ll} & wlp(Fish.setPace, B) \\ \equiv & wlp(\{this \in Fish\}\ ;\ \{p > 0\}\ ;\ this.d := p, \\ & \quad (\forall f \in Fish \bullet f.d > 0 \wedge 0 \leq f.x < W)) \\ \Leftarrow & this \in Fish \wedge p > 0 \Rightarrow \\ & \quad (\forall f \in Fish \bullet f.(d[this \leftarrow p]) > 0 \wedge 0 \leq f.x < W)) \\ \Leftarrow & (\forall f \in Fish \bullet f.d > 0 \wedge 0 \leq f.x < W) \\ \Leftarrow & B \end{array}$$

For (d.1) we proceed similarly, now applying (35):

$$\begin{array}{ll} & wlp(Fish.moveRight, B) \\ \equiv & wlp((\sqcap this \in Fish \bullet [this.x + this.d < W \wedge this.r]\ ;\ this.x := \\ & \quad this.x + this.d), (\forall f \in Fish \bullet f.d > 0 \wedge 0 \leq f.x < W)) \\ \Leftarrow & (\forall this \in Fish \bullet this.x + this.d < W \wedge this.r \Rightarrow \\ & \quad (\forall f \in Fish \bullet f.d > 0 \wedge 0 \leq f.(x[this \leftarrow this.x + this.d]) < W))) \\ \Leftarrow & (\forall f \in Fish \bullet f.d > 0 \wedge 0 \leq f.x < W) \\ \equiv & B \end{array}$$

The proof of (d.2) is similar and left out. Conditions (d.3) and (d.4) follow immediately as the corresponding actions to not change any variable mentioned in the invariant, hence preserve the invariant vacuously. In concluding with this example we note that B is a *local* invariant as it does not relate the attributes of different objects; it is a quantification of conditions ranging over a single object. The technique equally applies to *global* invariants. We also note that in invariance proofs we may make use of the (finite) *conjunctivity* of *wlp*, which follows from the (finite) conjunctivity of *wp* (as can be checked for each of the defined statements):

$$wp(S, b \wedge c) \equiv wp(S, b) \wedge wp(C, c) \tag{41}$$
$$wlp(S, b \wedge c) \equiv wlp(S, b) \wedge wlp(C, c) \tag{42}$$

4 Refinement

Class refinement builds on the notion of *data refinement* of statements. Ordinary (algorithmic) refinement of statement S by T, written $S \sqsubseteq T$ holds if for all predicates c, $wp(S, c) \Rightarrow wp(T, c)$. This implies that T can be used for whatever S can be, but T may be "more deterministic", may have a weaker termination domain, and may have a stronger guard. Data refinement $S \sqsubseteq_R T$ generalizes this by allowing S and T to operate on different variables, related through *coupling invariant* or *refinement invariant* R. Among the various ways, data refinement can be introduced through the *conjugate weakest precondition* predicate transformer $\overline{wp}$, defined as $\overline{wp}(S, c) \equiv \neg wp(S, \neg c)$ (see [12] for a proof of equivalence of various definitions). Intuitively, $\overline{wp}$ is like wp for assignments and sequential composition, but exchanges guards with assertions and exchanges demonic with angelic nondeterminism. In case all program expressions are defined we have:

$$\begin{array}{lll}
\overline{wp}(\{b\}, c) & \equiv b \Rightarrow c & \quad (43) \\
\overline{wp}([b], c) & \equiv b \wedge c & \quad (44) \\
\overline{wp}(x := e, c) & \equiv c[x \backslash e] & \quad (45) \\
\overline{wp}(S\ ;\ T, c) & \equiv \overline{wp}(S, \overline{wp}(T, c)) & \quad (46) \\
\overline{wp}(S \sqcap T, c) & \equiv \overline{wp}(S, c) \vee \overline{wp}(T, c) & \quad (47) \\
\overline{wp}(\sqcap x \in s \bullet S, c) & \equiv (\exists x \in s \bullet \overline{wp}(S, c)) \qquad x \text{ not free in } c & \quad (48)
\end{array}$$

In case program expressions are possibly undefined we have:

$$\begin{array}{lll}
\overline{wp}(\{b\}, c) & \equiv \Delta\, b \wedge b \Rightarrow c & \quad (49) \\
\overline{wp}([b], c) & \equiv \Delta\, b \Rightarrow b \wedge c & \quad (50) \\
\overline{wp}(x := e, c) & \equiv \Delta\, e \Rightarrow c[x \backslash e] & \quad (51)
\end{array}$$

Let S be a statement over variables s and T a statement over variables t, where s and t are disjoint. Let R be a predicate over s and t. Statement S is refined by T through R, written $S \sqsubseteq_R T$, is defined by:

$$S \sqsubseteq_R T \equiv R \wedge \mathit{trm}\, S \Rightarrow wp(T, \overline{wp}(S, R)) \tag{52}$$

In case S and T have variables r in common—say global variables or results—the definition needs to be extended. Let $S[x \backslash y]$ stand for statement S with variables x substituted by variables y. Assume that $\overline{r}$ are fresh variables:

$$S \sqsubseteq_R T \equiv R \wedge \mathit{trm}\, S \Rightarrow wp(T[r \backslash \overline{r}], \overline{wp}(S, R \wedge r = \overline{r})) \tag{53}$$

As a useful special case is the refinement of *skip*:

$$\mathit{skip} \sqsubseteq_R T \equiv R \Rightarrow wp(T, R) \tag{54}$$

Components of a sequential composition can be refined individually:

$$S_0 \sqsubseteq_R T_0 \wedge S_1 \sqsubseteq_R T_1 \Rightarrow S_0\ ;\ S_1 \sqsubseteq_R T_0\ ;\ T_1 \tag{55}$$

Definition 2 (Class Refinement) *Let C be a class with attributes c and D be a class with attributes d. We assume that both classes have the same method names and parameter and return types, and that each action defined in C is also defined in D. However, class D may have additional actions, called auxiliary actions, and referred to by $D.aux$. Let R be a predicate over c and d. Class C is refined by D through R, written $C \sqsubseteq_R D$, if following conditions hold:*

(a) Program Initialization: *When no objects exists, the refinement invariant holds:*

$$C = \{\} \wedge D = \{\} \Rightarrow R$$

(b) Object Creation: *The creation of a C object is refined by the creation of a D object:*

$$C.new \sqsubseteq_R D.new$$

(c) Method Refinement: *Every method meth of C is refined by the corresponding method in D:*

$$C.meth \sqsubseteq_R D.meth$$

Method Enabledness: *For every method meth in C, either the corresponding method of D or* some *action in D is enabled:*

$$R \wedge \mathit{grd}\ C.meth \wedge \mathit{trm}\ C.meth \Rightarrow \mathit{grd}\ D.meth \vee (\vee act \bullet \mathit{grd}\ D.act)$$

(d) Main Action Refinement: *Every action act of C is refined by the corresponding action in D:*

$$C.act \sqsubseteq_R D.act$$

Main Action Enabledness: *For every action act in C,* some *action in D is enabled:*

$$R \wedge \mathit{grd}\ C.act \wedge \mathit{trm}\ C.act \Rightarrow (\vee act \bullet \mathit{grd}\ D.act)$$

(e) Auxiliary Action Refinement: *Every new action aux of D refines skip:*

$$skip \sqsubseteq_R D.aux$$

Auxiliary Action Termination: *The computation of auxiliary actions terminates eventually:*

$$R \Rightarrow \textit{all actions } D.aux \textit{ terminate eventually}$$

Condition (b) on object creation does not include a check for enabledness, like condition (c) does, as we assume that initializations are always enabled: the syntactic structure of initializations does not allow for guards. Condition (b) can be simplified by noting that the refinement invariant has to satisfy a *healthiness condition*, namely that for every C object there must exist at least one D object with the same identity:

$$R \Rightarrow C \subseteq D \tag{56}$$

Predicate R may imply an exact one-to-one correspondence $C = D$, as in the delayed vector summation below, or may allow for more D objects than C objects, as would be for the refinement of *PriorityBag* by *PriorityQueue*. The necessity for this healthiness condition can be seen by expanding and simplifying condition (b) to $\mathit{this} :\notin C \cup \{\mathit{nil}\} \ ; \ C := C \cup \{\mathit{this}\} \ ; \ C.\mathit{init} \sqsubseteq_R \mathit{this} :\notin D \cup \{\mathit{nil}\} \ ; \ D := D \cup \{\mathit{this}\} \ ; \ D.\mathit{init}$. Assuming that initializations do not assign to *this*, which typically would be syntactically forbidden, for above to hold, $\mathit{this} :\notin C \cup \{\mathit{nil}\} \sqsubseteq_R \mathit{this} :\notin D \cup \{\mathit{nil}\}$ has already to hold, as the subsequent statements cannot possibly establish R otherwise. We calculate:

$$\begin{aligned}
& \mathit{this} :\notin C \cup \{\mathit{nil}\} \sqsubseteq_R \mathit{this} :\notin D \cup \{\mathit{nil}\} \\
\equiv\ & R \Rightarrow \mathit{wp}(\overline{\mathit{this}} :\notin D \cup \{\mathit{nil}\}, \overline{\mathit{wp}}(\mathit{this} :\notin C \cup \{\mathit{nil}\}, R \land \mathit{this} = \overline{\mathit{this}})) \\
\equiv\ & R \Rightarrow (\forall \overline{\mathit{this}} \notin D \cup \{\mathit{nil}\} \bullet (\exists \mathit{this} \notin C \cup \{\mathit{nil}\} \bullet R \land \mathit{this} = \overline{\mathit{this}})) \\
\equiv\ & R \Rightarrow (\forall \overline{\mathit{this}} \notin D \cup \{\mathit{nil}\} \bullet \overline{\mathit{this}} \notin C \cup \{\mathit{nil}\}) \\
\equiv\ & R \Rightarrow C \subseteq D
\end{aligned}$$

From this observation and (55) we can immediately derive an alternative formulation of condition (b):

(b') *Object Creation:* Provided $R \Rightarrow C \subseteq D$

$$C := C \cup \{\mathit{this}\} \ ; \ C.\mathit{init} \sqsubseteq_R D := D \cup \{\mathit{this}\} \ ; \ D.\mathit{init}$$

Condition (e) implies that the auxiliary actions are *stuttering* actions: as they refine *skip*, their effect is not visible from C and as they eventually terminate, they do not introduce (observable) non-termination. The second part of (b) can in general be shown with the use of a *variant* t, an integer expression. The conditions are that for all new actions $D.\mathit{aux}$, if $D.\mathit{aux}$ is enabled, then t must be strictly greater than 0 and $D.\mathit{aux}$ decreases t. Let v be an auxiliary variable:

> *Auxiliary Action Termination:* All auxiliary actions decrease t and become disabled if t reaches 0:
>
> $$\begin{aligned} R \land \mathit{grd}\ D.\mathit{aux} &\Rightarrow t > 0 \\ R \land t = v &\Rightarrow \mathit{wp}(D.\mathit{aux}, t < v) \end{aligned}$$

This definition of class refinement is justified by appealing to the refinement of action systems with procedures, as done in [7, 8, 10]. The difference to these approaches is the treatment of object identities, here we follow [17]. We note that class refinement can be further generalized if needed [10]: abstract stuttering can be allowed to be removed in the refinement, and concrete stuttering actions can be more general than a refinement of *skip*.

We give an example of a delayed vector summation that illustrates the concept of delaying a computation by enabling a background action. The example makes use of arrays. If a is declared as **array** N **of** T, we understand a to be a function and define the array update statement as a function update:

$$a(e) := f = a := a[e \leftarrow f]$$

As indexing an array out of bounds is an error, we need to specify the definedness of program expressions with array access accordingly:

$$\begin{aligned}
\Delta(a(e)) &\equiv \Delta\, e \wedge 0 \leq e < N \\
\Delta(a[e \leftarrow f]) &\equiv \Delta\, e \wedge \Delta\, f \wedge 0 \leq e < N
\end{aligned}$$

Class $V0$ allows to store elements of a vector and their sum to be calculated. Class $V1$ performs the summation in the background and blocks the request for the sum if it is not yet calculated:

class $V0$
  var a : **array** M **of** *integer*
  var s : *integer*
  method $set(j, e : integer)$
    $a(j) := e$
  method *calcSum*
    $s := (\sum j \mid 0 \leq j < M \bullet a(j))$
  method *getSum* : *integer*
    $result := s$
end

class $V1$
  var a : **array** M **of** *integer*
  var s, m : *integer*
  initialization $m := 0$
  method $set(j, e : integer)$
    when $j \geq m$ **do** $a(j) := e$
  method *calcSum*
    $s, m := 0, M$
  method *getSum* : *integer*
    when $m = 0$ **do** $result := s$
  action *addElt*
    when $m > 0$ **do**
      $s, m := s + a(m-1), m-1$
end

Thus the conditions for $V0$ to be a refined by $V1$ through R are:

(a) $V0 = \{\} \wedge V1 = \{\} \Rightarrow R$
(b) $V0.new \sqsubseteq_R V1.new$
(c.1) $V0.set \sqsubseteq_R V1.set$
  $R \wedge grd\ V0.set \wedge trm\ V0.set \Rightarrow grd\ V1.set \vee grd\ V1.addElt$
(c.2) $V0.calcSum \sqsubseteq_R V1.calcSum$
  $R \wedge grd\ V0.calcSum \wedge trm\ V0.calcSum \Rightarrow$
    $grd\ V1.calcSum \vee grd\ V1.addElt$

(c.3) $V0.getSum \sqsubseteq_R V1.getSum$
$R \land grd\ V0.getSum \land trm\ V0.getSum \Rightarrow$
$grd\ V1.getSum \lor grd\ V1.addElt$
(e) $skip \sqsubseteq_R V1.addElt$
$R \Rightarrow$ action $V1.addElt$ terminates eventually

Abbreviating $V0.a$, $V0.s$, $V1.a$, $V1.s$, $V1.m$ by a_0, s_0, a_1, s_1, m, we use as the refinement invariant:

$$\begin{array}{l} R \equiv V0 = V1 \land \\ \quad (\forall v \in V0 \bullet v.a_0 = v.a_1 \land 0 \leq v.m \leq M \land \\ \quad\quad v.s_0 = v.s_1 + (\sum j \mid 0 \leq j < v.m \bullet v.a_0(j)) \end{array}$$

Condition (a) amounts to a quantification over an empty range, which holds vacuously. For (b) it is sufficient to use the condition (b') instead. We have that $V0.init = this.a_0, this.s_0 := ?, ?$, hence $trm(V0 := V0 \cup \{this\} \ ; \ V0.init) \equiv true$, and $V1.init = this.m := 0 \ ; \ this.a1, this.s1 := ?, ?$. In the proof, we first apply the definition of $\sqsubseteq_R$, then the rules of wp and $\overline{wp}$, then perform the substitution, and finally simplify the outcome by a case analysis with $v = this$ and $v \neq this$:

$$\begin{array}{l} \quad V0 := V0 \cup \{this\} \ ; \ V0.init \sqsubseteq_R V1 := V1 \cup \{this\} \ ; \ V1.init \\ \equiv R \Rightarrow wp(V1 := V1 \cup \{this\} \ ; \ V1.init, \\ \quad\quad \overline{wp}(V0 := V0 \cup \{this\} \ ; \ V0.init, R)) \\ \equiv R \Rightarrow (\forall g_1, h_1 \bullet \exists g_0, h_0 \bullet \\ \quad\quad R[a_0, s_0 \backslash a_0[this \leftarrow h_0], s_0[this \leftarrow g_0][V0 \backslash V0 \cup \{this\}] \\ \quad\quad [a_1, s_1 \backslash a_1[this \leftarrow h_1], s_1[this \leftarrow g_1][m \backslash m[this \leftarrow 0][V1 \backslash V1 \cup \{this\}]) \\ \equiv R \Rightarrow (\forall g_1, h_1 \bullet \exists g_0, h_0 \bullet \ V0 \cup \{this\} = V1 \cup \{this\} \land \\ \quad\quad (\forall v \in V0 \bullet v.a_0[this \leftarrow h_0] = v.a_1[this \leftarrow h_1] \land \\ \quad\quad\quad 0 \leq v.m[this \leftarrow 0] \leq M \land \\ \quad\quad\quad v.s_0[this \leftarrow g_0] = v.s_1[this \leftarrow g_1] + \\ \quad\quad\quad\quad (\sum j \mid 0 \leq j < v.m[this \leftarrow 0] \bullet a_0[this \leftarrow h_0](j)))) \\ \equiv true \end{array}$$

For the first part of (c.1) we have that $V0.set = \{this \in V0\} \ ; \ this.a_0(j) := e$ and $V1.set = \{this \in V1\} \ ; \ [j \geq this.m] \ ; \ this.a_1(j) := e$. With (27), (28), (15) we get that $trm\ V0.set \equiv this \in V0 \land 0 \leq j < M$:

$$\begin{array}{l} \quad V0.set \sqsubseteq_R V1.set \\ \equiv R \land trm\ V0.set \Rightarrow wp(V1.set, \overline{wp}(V2.set, R)) \\ \equiv R \land this \in V0 \land 0 \leq j < M \Rightarrow \\ \quad\quad wp(\{this \in V1\} \ ; \ [j \geq this.m] \ ; \ this.a_1(j) := e, \\ \quad\quad\quad \overline{wp}(\{this \in V0\} \ ; \ this.a_0(j) := e, R) \\ \equiv R \land this \in V0 \land 0 \leq j < M \Rightarrow this \in V1 \land (j \geq this.m \Rightarrow \\ \quad\quad (this \in V0 \land R[a_0 \backslash a_0[this \leftarrow this.a_0[j \leftarrow e]]]) \\ \quad\quad\quad [a_1 \backslash a_1[this \leftarrow this.a_1[j \leftarrow e]]]) \\ \equiv true \end{array}$$

For the second part of (c.1) we first observe that $V1.addElt = (\sqcap this \in V1 \bullet [this.m > 0]\ ;\ this.s_1, this.m := this.s_1 + this.a_1(this.m-1), this.m-1)$. With (26) and (25) we get that $grd\ V0.set \equiv true$, $grd\ V1.set \equiv j \geq this.m$ and that $grd\ V1.addElt \equiv (\exists this \in V1 \bullet this.m > 0)$:

$$
\begin{aligned}
& R \wedge grd\ V0.set \wedge trm\ V0.set \Rightarrow grd\ V1.set \vee grd\ V1.addElt \\
\equiv\ & R \wedge this \in V0 \wedge 0 \leq j < M \Rightarrow j \geq this.m \vee (\exists this \in V1 \bullet this.m > 0) \\
\Leftarrow\ & R \wedge this \in V0 \wedge 0 \leq j < M \Rightarrow j \geq this.m \vee this.m > 0 \\
\equiv\ & true
\end{aligned}
$$

The conditions (c.2) and (c.3) can be discharged similarly. For the first part of condition (e) we apply (54), then (7), (3), (19), (15), perform the substitutions, and finally simplify the outcome by a case analysis with $v = this$ and $v \neq this$:

$$
\begin{aligned}
& skip \sqsubseteq_R V1.addElt \\
\equiv\ & R \Rightarrow wp(V1.addElt, R) \\
\equiv\ & R \Rightarrow wp((\sqcap this \in V1 \bullet [this.m > 0]\ ; \\
& \quad this.s_1, this.m := this.s_1 + this.a_1(this.m-1), this.m-1), R) \\
\equiv\ & R \Rightarrow (\forall this \in V1 \bullet this.m > 0 \Rightarrow 0 \leq this.m - 1 < M \wedge \\
& R[s_1, m \backslash s_1[this \leftarrow this.s_1 + this.a_1(this.m-1)], m[this \leftarrow this.m-1]) \\
\equiv\ & R \Rightarrow (\forall this \in V1 \bullet this.m > 0 \Rightarrow 0 \leq this.m - 1 < M \wedge \\
& \quad V0 = V1 \wedge \\
& \quad (\forall v \in V0 \bullet v.a_0 = v.a_1 \wedge 0 \leq v.m[this \leftarrow this.m-1] \leq M \wedge \\
& \qquad v.s_0 = v.s_1[this \leftarrow this.s_1 + this.a_1(this.m-1)] + \\
& \qquad\quad (\textstyle\sum j \mid 0 \leq j < v.m[this \leftarrow this.m-1] \bullet v.a_0(j)))) \\
\equiv\ & true
\end{aligned}
$$

For the second part of condition (e) we use $(\sum v \in V1 \bullet v.m)$ as the variant and get following two conditions:

$$
\begin{aligned}
& R \wedge grd\ V1.addElt && \Rightarrow (\textstyle\sum j \in V1 \bullet v.m) > 0 \\
& R \wedge (\textstyle\sum v \in V1 \bullet v.m) = w && \Rightarrow wp(V1.addElt, (\textstyle\sum v \in V1 \bullet v.m) < w)
\end{aligned}
$$

Again, these conditions can be discharged with the given rules. We omit the proofs, but like to stress the inherent structure of the conditions, as exemplified with the last one: the refinement invariant and the variant range over *all* objects of a class, not just a single object. This allows the refinement to span several objects if needed, as would be the case with the refinement of *PriorityBag* by *PriorityQueue*.

5 Non-Atomic Actions

For verifying non-atomic operations, these need first to be transformed into the kernel language of atomic operations. Suppose that an action A is of the form $S\ ;\ T$, where S and T are atomic. We can make the *atomic regions* explicit by including them in atomicity brackets and writing $A = \langle S\rangle\ ;\ \langle T\rangle$. Such an action needs to be split into two actions, an action $A0$ that executes only S and enables T and an action $A1$ that executes only T. As an action may be initiated multiple times, a counter, say c, for recording the invocations of S is needed. Thus the transformation results in $A0 = S\ ;\ c := c+1$ and $A1 = [c > 0]\ ;\ c := c-1\ ;\ T$. The counter c has to be made an attribute of the corresponding class and to be initialized to 0. When local variables are present, a simple counter is not sufficient. Suppose we have $A = \langle \mathbf{var}\ x\ ;\ \mathbf{begin}\ S\rangle\ ;\ \langle T\ \mathbf{end}\rangle$. If A is initiated multiple times, multiple copies of x would exist. They are there stored in bag, say b. The transformation would then result in $A0 = \mathbf{var}\ x\ ;\ \mathbf{begin}\ S\ ;\ b := b + [x]\ \mathbf{end}$ and $A1 = \mathbf{var}\ x \in b\ ;\ \mathbf{begin}\ b := b - [x]\ ;\ T\ \mathbf{end}$. Here x is in general a tuple (list) of variables, and b is a bag of tuples that is made an attribute of the corresponding class.

Local variables necessarily arise with method calls. According to (33) an atomic call $x := c.\mathit{meth}(e)$ gives rise to local variables for copies of c, e, and the result x. Indeed, in our implementation first x and e are evaluated before a call to $c.\mathit{meth}$ is attempted and other operations cannot affect these values. Thus, even a "parameterless" method call requires at least the receiver of the call to be stored in a bag.

Rather than formalizing this transformation itself, we illustrate it with the example of dining philosophers. We use the same syntax for non-atomic and atomic methods and actions, except that we may chose to make the atomic regions explicit.

As an example, we like to show *mutual exclusion* of philosophers, in the sense that no two philosophers sharing a fork can eat at the same time. We do so for an arbitrary arrangement of philosophers and forks, not just for a circular one. First, class *Phil* is rewritten to explicitly indicate the atomic regions by atomicity brackets; in class *Fork* all methods are atomic. Some atomic regions are labeled:

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Phil}\\
\quad \mathbf{var}\ \mathit{state} : (\mathit{thinking}, \mathit{hungry}, \mathit{eating}, \mathit{full})\\
\quad \mathbf{var}\ \mathit{left}, \mathit{right} : \mathit{Fork}\\
\quad \mathbf{initialization}\ (l, r : \mathit{Fork})\\
\quad\quad \langle \mathit{state}, \mathit{left}, \mathit{right} := \mathit{thinking}, l, r\rangle\\
\quad \mathbf{action}\ \mathit{needToEat}\\
\quad\quad \langle \mathbf{when}\ \mathit{state} = \mathit{thinking}\ \mathbf{do}\\
\quad\quad\quad \mathbf{begin}\ \mathit{state} := \mathit{hungry}\ ;\\
\quad\quad\quad\quad \mathbf{var}\ \mathit{this}\ ;\ \mathbf{begin}\ \mathit{this} := \mathit{left}\ ;\ \rangle\\
\quad\quad \mathrm{at1:}\quad \mathit{Fork.pickUp}\\
\quad\quad \mathrm{at2:}\ \langle \mathbf{end}\ ;
\end{array}
$$

```
            var this ; begin this := right ; ⟩
        at3:    Fork.pickUp
        at4: ⟨end ;
              state := eating
            end⟩
    action needToThink
      ⟨when state = eating do
          begin state := full ;
            var this ; begin this := left ; ⟩
        at5:    Fork.putDown
        at6: ⟨end ;
            var this ; begin this := right ; ⟩
        at7:    Fork.putDown
        at8: ⟨end ;
              state := thinking
            end⟩
  end
```

Transforming every atomic region into an action and adding the counters results in:

```
  class Phil
    var state : (thinking, hungry, eating, full)
    var left, right : Fork
    var at1, at3, at5, at7 : bag of Object
    var at2, at4, at6, at8 : natural
    initialization (l, r : Fork)
      state, left, right, at1, at2, at3, at4, at5, at6, at7, at8 :=
        thinking, l, r, [], 0, [], 0, [], 0, [], 0
    action needToEat0
      when state = thinking do
        begin state := hungry ; at1 := at1 + [left] end
    action needToEat1
      var this ∈ at1 ;
        begin at1 := at1 − [this] ; Fork.pickUp ; at2 := at2 + 1 end
    action needToEat2
      when at2 > 0 do begin at2 := at2 − 1 ; at3 := at3 + [right] end
    action needToEat3
      var this ∈ at3 ;
        begin at3 := at3 − [this] ; Fork.pickUp ; at4 := at4 + 1 end
    action needToEat4
      when at4 > 0 do
        begin at4 := at4 − 1 ; state := eating end
    action needToThink0
      when state = eating do
```

$$
\begin{array}{l}
\qquad\qquad \textbf{begin } \mathit{state} := \mathit{full}\ ;\ \mathit{at}5 := \mathit{at}5 + [\mathit{left}]\ \textbf{end} \\
\quad \textbf{action } \mathit{needToThink}1 \\
\qquad \textbf{var } \mathit{this} \in \mathit{at}5\ ; \\
\qquad\qquad \textbf{begin } \mathit{at}5 := \mathit{at}5 - [\mathit{this}]\ ;\ \mathit{Fork.putDown}\ ;\ \mathit{at}6 := \mathit{at}6 + 1\ \textbf{end} \\
\quad \textbf{action } \mathit{needToThink}2 \\
\qquad \textbf{when } \mathit{at}6 > 0 \textbf{ do begin } \mathit{at}6 := \mathit{at}6 - 1\ ;\ \mathit{at}7 := \mathit{at}7 + [\mathit{right}]\ \textbf{end} \\
\quad \textbf{action } \mathit{needToThink}3 \\
\qquad \textbf{var } \mathit{this} \in \mathit{at}7\ ; \\
\qquad\qquad \textbf{begin } \mathit{at}7 := \mathit{at}7 - [\mathit{this}]\ ;\ \mathit{Fork.putDown}\ ;\ \mathit{at}8 := \mathit{at}8 + 1\ \textbf{end} \\
\quad \textbf{action } \mathit{needToThink}4 \\
\qquad \textbf{when } \mathit{at}8 > 0 \textbf{ do} \\
\qquad\qquad \textbf{begin } \mathit{at}8 := \mathit{at}8 - 1\ ;\ \mathit{state} := \mathit{eating}\ \textbf{end} \\
\textbf{end}
\end{array}
$$

Predicate $ne(Ph, f, state)$ is defined to mean that all philosophers of the set Ph who are sharing fork f are not in the state of *eating*:

$$
\begin{array}{l}
ne(Ph, f, state) \equiv (\forall ph \in Ph \bullet (ph.left = f \lor ph.right = f) \Rightarrow \\
\qquad\qquad\qquad\qquad ph.state \neq eating)
\end{array}
$$

The mutual exclusion property is expressed as:

$$
\begin{array}{l}
X \equiv (\forall ph \in Phil \bullet ph.state = eating \Rightarrow \\
\qquad\quad ne(Phil - \{ph\}, ph.left, state) \land ne(Phil - \{ph\}, ph.right, state))
\end{array}
$$

Instead of showing that X is an invariant, we have to show a stronger condition. It is constructed as follows. First, if a fork is available, then no philosopher sharing that fork can be eating:

$$
FR \equiv (\forall f \in Fork \bullet f.available \Rightarrow ne(Phil, f, state))
$$

Second, for all eating philosophers, both their left and right fork are not available and no other philosopher who is sharing one of these forks can be eating:

$$
\begin{array}{l}
PH \equiv (\forall ph \in Phil \bullet ph.state = eating \Rightarrow \\
\qquad\quad \neg ph.left.available \land ne(Phil - \{ph\}, ph.left, state) \land \\
\qquad\quad \neg ph.right.available \land ne(Phil - \{ph\}, ph.right, state))
\end{array}
$$

Third, we specify the enabledness of the atomic actions. Let $\#b$ stand for the number of elements in bag b.

$$
\begin{array}{l}
EA \equiv \\
\quad (\forall ph \in Phil \bullet \\
\qquad 0 \leq \#ph.at1 + ph.at2 + \#ph.at3 + ph.at4 + \\
\qquad\quad \#ph.at5 + ph.at6 + \#ph.at7 + ph.at8 \leq 1 \land \\
\qquad (\#ph.at1 + ph.at2 + \#ph.at3 + ph.at4 > 0 \equiv ph.state = hungry) \land \\
\qquad (\#ph.at5 + ph.at6 + \#ph.at7 + ph.at8 > 0 \equiv ph.state = full))
\end{array}
$$

Finally, we specify the "intermediate assertions":

$$
\begin{aligned}
&AT1 \equiv (\forall ph \in \mathit{Phil} \bullet \#ph.at1 > 0 \Rightarrow ph.at1 = [ph.\mathit{left}])\\
&AT2 \equiv (\forall ph \in \mathit{Phil} \bullet ph.at2 > 0 \Rightarrow\\
&\qquad\qquad \neg ph.\mathit{left}.\mathit{available} \wedge ne(\mathit{Phil} - \{ph\}, ph.\mathit{left}, \mathit{state})\\
&AT3 \equiv (\forall ph \in \mathit{Phil} \bullet \#ph.at3 > 0 \Rightarrow ph.at3 = [ph.\mathit{right}] \wedge\\
&\qquad\qquad \neg ph.\mathit{left}.\mathit{available} \wedge ne(\mathit{Phil} - \{ph\}, ph.\mathit{left}, \mathit{state})\\
&AT4 \equiv (\forall ph \in \mathit{Phil} \bullet ph.at4 > 0 \Rightarrow\\
&\qquad\qquad \neg ph.\mathit{left}.\mathit{available} \wedge ne(\mathit{Phil} - \{ph\}, ph.\mathit{left}, \mathit{state})\\
&\qquad\qquad \neg ph.\mathit{right}.\mathit{available} \wedge ne(\mathit{Phil} - \{ph\}, ph.\mathit{left}, \mathit{state})\\
&AT5 \equiv (\forall ph \in \mathit{Phil} \bullet \#ph.at5 > 0 \Rightarrow ph.at5 = [ph.\mathit{left}] \wedge\\
&\qquad\qquad \neg ph.\mathit{left}.\mathit{available} \wedge ne(\mathit{Phil} - \{ph\}, ph.\mathit{left}, \mathit{state}) \wedge\\
&\qquad\qquad \neg ph.\mathit{right}.\mathit{available} \wedge ne(\mathit{Phil} - \{ph\}, ph.\mathit{right}, \mathit{state})\\
&AT6 \equiv (\forall ph \in \mathit{Phil} \bullet ph.at6 > 0 \Rightarrow\\
&\qquad\qquad \neg ph.\mathit{right}.\mathit{available} \wedge ne(\mathit{Phil} - \{ph\}, ph.\mathit{right}, \mathit{state})\\
&AT7 \equiv (\forall ph \in \mathit{Phil} \bullet \#ph.at7 > 0 \Rightarrow ph.at7 = [ph.\mathit{right}]\\
&\qquad\qquad \neg ph.\mathit{right}.\mathit{available} \wedge ne(\mathit{Phil} - \{ph\}, ph.\mathit{right}, \mathit{state})
\end{aligned}
$$

The claimed invariant, E, is constructed as the conjunction of all the above conditions:

$$E \equiv \mathit{FR} \wedge \mathit{PH} \wedge \mathit{EA} \wedge AT1 \wedge AT2 \wedge AT3 \wedge AT4 \wedge AT5 \wedge AT6 \wedge AT7$$

As already PH implies X, the mutual exclusion condition X follows from E being an invariant, which holds if:

(a) $\mathit{Phil} = \{\} \Rightarrow E$
(b) $E \Rightarrow wlp(\mathit{Phil}.\mathit{new}, E)$
(d.1) $E \Rightarrow wlp(\mathit{Phil}.\mathit{needToEat}0, E)$
(d.2) $E \Rightarrow wlp(\mathit{Phil}.\mathit{needToEat}1, E)$
(d.3) $E \Rightarrow wlp(\mathit{Phil}.\mathit{needToEat}2, E)$
(d.4) $E \Rightarrow wlp(\mathit{Phil}.\mathit{needToEat}3, E)$
(d.5) $E \Rightarrow wlp(\mathit{Phil}.\mathit{needToEat}4, E)$
(d.6) $E \Rightarrow wlp(\mathit{Phil}.\mathit{needToThink}0, E)$
(d.7) $E \Rightarrow wlp(\mathit{Phil}.\mathit{needToThink}1, E)$
(d.8) $E \Rightarrow wlp(\mathit{Phil}.\mathit{needToThink}2, E)$
(d.9) $E \Rightarrow wlp(\mathit{Phil}.\mathit{needToThink}3, E)$
(d.10) $E \Rightarrow wlp(\mathit{Phil}.\mathit{needToThink}4, E)$

Condition (a) amounts to quantifications over empty sets, which all hold vacuously. For (b) we have:

$$
\begin{aligned}
&\mathit{Phil}.\mathit{new} =\\
&\quad \mathit{this} :\notin \mathit{Phil} \cup \{\mathit{nil}\}\ ;\ \mathit{Phil} := \mathit{Phil} \cup \{\mathit{this}\}\ ;\ \mathit{this}.\mathit{state}, \mathit{this}.\mathit{left},\\
&\qquad \mathit{this}.\mathit{right}, \mathit{this}.at1, \mathit{this}.at2, \mathit{this}.at3, \mathit{this}.at4, \mathit{this}.at5, \mathit{this}.at6,\\
&\qquad\quad \mathit{this}.at7, \mathit{this}.at8 := \mathit{thinking}, l, r, [], 0, [], 0, [], 0, [], 0
\end{aligned}
$$

By (42) we consider the postconditions FR, PH, EA, $AT1$, $AT2$, $AT3$ $AT4$, $AT5$, $AT6$, and $AT7$ in turn. For FR we consider the cases $ph = this$ and $ph \neq this$ in the quantification of ne: if $ph = this$, then, as $this.state$ is set to $thinking$, the conclusion of the implication is true and the whole predicate is true. If $ph \neq this$, then this case follows from the precondition FR. For PH we make the same case analysis with $ph = this$ and $ph \neq this$, and note that $this.state$ is set to $thinking$, so the hypothesis of the implication for that case is false and the whole implication becomes true. For EA we make the same case analysis and note that for $ph = this$, all of $this.at1, \ldots, this.at6$ are set to 0 and $this.state$ is set to $thinking$, so the whole predicate becomes true in that case. For $AT2$ to $AT7$ we have that the hypotheses are all false in the case of $ph = this$, so these are preserved as well. For condition (d.1) we have:

$$
\begin{array}{l}
Phil.needToEat0 = \\
\quad (\sqcap this \in Phil \bullet [this.state = thinking] \ ; \ this.state := hungry \ ; \\
\qquad this.at1 := this.at1 + [this.left])
\end{array}
$$

For postcondition FR we consider the cases $ph = this$ and $ph \neq this$ in the quantification of ne: if $ph = this$, then, as $this.state$ is set to $hungry$, the conclusion of the implication is true and the whole predicate becomes true. If $ph \neq this$, then this case follows from the precondition FR. For postcondition PH we make the same case analysis with $ph = this$ and $ph \neq this$, and note that $this.state$ is set to $hungry$, so the hypothesis of the implication for that case is false and the whole implication becomes true. For postcondition EA we make the same case analysis and note that for $ph = this$, from the precondition EA and the guard $this.state = thinking$, we know that initially all of $this.at1$, $\ldots$, $this.at8$ are $[]$ or 0 , hence finally $\#this.at1$ is 1 and EA is preserved. Postcondition $AT1$ is established by the assignment to $this.at1$, as the guard $this.state = thinking$ and the precondition EA together imply that $this.at1$ is empty initially. For postconditions $AT2$ to $AT7$ we have that the hypotheses are all false in the case of $ph = this$, so these are preserved as well, concluding the proof of (d.1). For condition (d.2) we have, after renaming:

$$
\begin{array}{l}
Phil.needToEat1 = \\
\quad (\sqcap this \in Phil \bullet \sqcap \overline{this} \in this.at1 \bullet this.at1 := this.at1 - [\overline{this}] \ ; \\
\qquad [\overline{this}.available] \ ; \ \overline{this}.available := false \ ; \ this.at2 := this.at2 + 1)
\end{array}
$$

For postcondition FR we observe that, as $f.available$ is set to $false$ for some f, the implication becomes true and FR is preserved. For postcondition PH we note that as $this.left.available$ is set of $false$ and all other variables of PH are unchanged, PH cannot be invalidated. For postcondition EA we make a case analysis and note that for $ph = this$, the sum of $\#this.at1$ and $this.at2$ remains unchanged, so the EA is preserved as well. Postcondition $AT1$ cannot be invalidated as $this.at1$ becomes empty. For postcondition $AT2$, in the case of $ph = this$, we note that $this.left.available$ is set to $false$ and that $ne(Phil - \{ph\}, ph.left, state)$ follows from the guard $this.left.available$ and precondition

FK. For postconditions $AT3$ to $AT8$ we have that the hypotheses are all false in the case of $ph = this$, so these are preserved as well.

Conditions (d.3) to (d.8) can be discharged analogously and are omitted here. In concluding this example we note that, as E spans objects of class *Fork* as well, calls to methods of *Fork* may invalidate E. That is, in order to show that E is an invariant of the whole program, we would need additionally to show that E is preserved by all other classes as well (which is easy to establish if other classes only create *Fork* objects and do not call *pickUp* and *putDown*).

6 Conclusions

We note that for our implementation, the object structure effectively helps to control the evaluation of guards. All guards must mention only attributes of the object itself. Without such a syntactic constraint, the guarded statement **when** *cond* **do** *stat* would require repeated evaluation of *cond* after some delay. To reduce resource contention, a *binary exponential back-off protocol* could be employed that starts with a random delay and doubles it after each failure. In the present implementation, no delays are employed. A number of threads in a thread pool are maintained and action guards are initially evaluated once when a thread is searching for an action to execute. Method guards are initially evaluated once when a method is called. Both action and method guards are reevaluated only after another thread has left the object and thus possibly affected the guards. We hope this measurements will show our implementation to be highly efficient.

While action system refinement [6, 18] appears as an attractive foundation for class refinement [7, 8, 10], under the assumption of atomicity the refinement rule is sound only if there is a single method call per action and method. For example, if $v \in V0$, then the sequence $v.calcSum$; $r := v.getSum$ would assign the sum to r, but if $v \in V1$ the sequence would always block as $v.calcSum$ disables $v.getSum$, even though $V0$ is refined by $V1$. Our approach is to ensure atomicity only up to method calls. As we furthermore allow multiple operations in one object to be initiated, but only one to progress, this leads to a disciplined form of inter-object concurrency. However, this fine-grained concurrency comes at a price:

First, additional attributes—counters and bags for local variables—need to be introduced. These make the class invariants and refinement invariants more complex than one would expect. The dining philosopher example shows how all intermediate assertions between atomic regions are captured by a single class invariant. It is not immediate how a Gries-Owicki style of reasoning with intermediate assertions could be applied in order to reduce the complexity of invariants. This may impose a limit on the practicability of the approach.

Second, while non-atomic actions can be dealt with, non-atomic methods cause problems: a method m defined as the sequential composition of two atomic statements S and T cannot be translated as a method m defined as S and an action for T, as a call to m would return without waiting for T to complete. A solution to this would be to replace a method call by its body, if needed repeatedly, and only then to apply the translation. However, this disallows recursive method calls. Developing a model that includes recursive calls is left as future work.

Acknowledgement. The author would like to thank the reviewers for their critical comments. They lead to substantial revision.

References

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* Series in Artificial Intelligence. MIT Press, Cambridge, MA, 1986.

[2] Pierre America. Pool-T: A parallel object-oriented language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, Computer Systems Series. MIT Press, Cambridge, MA, 1987.

[3] Pierre America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[4] Pierre America and Frank de Boer. Reasoning about dynamically evolving process structures. *Formal Aspects of Computing*, 6(3):269–316, 1994.

[5] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addision-Wesley, 2000.

[6] Ralph-Johan R. 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.

[7] Marcello. M. Bonsangue, Joost N. Kok, and Kaisa Sere. An approach to object-orientation in action systems. In J. Jeuring, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science 1422, pages 68–95, Marstrand, Sweden, June 1998, 1998. Springer-Verlag.

[8] 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.

[9] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency

and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, 1998.

[10] Martin Büchi and Emil Sekerinski. A foundation for refining concurrent objects. *Fundamenta Informaticae*, 44(1):25–61, 2000.

[11] Peter A. Buhr, Michel Fortier, and Michael H. Coffin. Monitor classification. *ACM Computing Surveys*, 27(1):63–107, 1995.

[12] Wei Chen and Jan Tijmen Udding. Towards a calculus of data refinement. In J. L. A. van de Snepscheut, editor, *Mathematics of Program Construction, 375th Anniversary of the Groningen University*, Lecture Notes in Computer Science 375, pages 197–218, Groningen, The Netherlands, 1989. Springer-Verlag.

[13] Cliff B. Jones. An object-based design method for concurrent programs. Technical report, University of Manchester, Department of Computer Science, December 1992.

[14] 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.

[15] Kevin Lou. *A Compiler for an Action-Based Object-Oriented Programming Language*. Master's thesis, McMaster University, 2004.

[16] Jayadev Misra. A simple, object-based view of multiprogramming. *Formal Methods in System Design*, 20(1):23–45, 2002.

[17] Emil Sekerinski. Concurrent object-oriented programs: From specification to code. In F. S. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects, First International Symposium, FMCO 02*, Lecture Notes in Computer Science 2852, pages 403–423, Leiden, The Netherlands, 2003. Springer-Verlag.

[18] Kaisa Sere and Marina Waldén. Data refinement of remote procedures. *Formal Aspects of Computing*, 12(4):278–297, 2000.

[19] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *ACM Conference on Object Oriented Programming Systems, Languages and Applications*, ACM SIGPLAN Notices, Vol 21, No 11, pages 258–268, 1986.