Under consideration for publication in Formal Aspects of Computing

Beyond Contracts for Concurrency

Jonathan S. Ostroff[1], Faraz Ahmadi Torshizi[1], Hai Feng Huang[1] and Bernd Schoeller[2]

[1]Dept. of Computer Science and Engineering,
York University,
4700 Keele Street,
Toronto, Canada M3J 1P3
[2] Software Engineering
ETH Zurich
Clausiusstrasse 59
CH-8092 Zurich

Abstract. SCOOP is a concurrent programming language with a new semantics for contracts that applies equally well in concurrent and sequential contexts. SCOOP eliminates race conditions and atomicity violations by construction. However, it is still vulnerable to deadlocks. In this paper we describe how far contracts can take us in verifying interesting properties of concurrent system using modular Hoare rules and show how theorem proving methods developed for sequential Eiffel can be extended to the concurrent case. However, some safety and liveness properties depend upon the environment and cannot be proved using the Hoare rules. To deal with such system properties, we outline a SCOOP Virtual Machine (SVM) as a fair transition system. The SVM makes it feasible to use modelchecking and theorem proving methods for checking global temporal logic properties of SCOOP programs. The SVM uses the Hoare rules where applicable to reduce the number of steps in a computation.

Keywords: SCOOP (Simple Concurrent Object Oriented programming), Concurrent Contracts, Operational Semantics, Verification, Temporal Logic.

1. Introduction

Sequential programming has a widely accepted set of ideas such as standard control and data structuring techniques, modularity, and information hiding constructs that now appear in object-oriented languages such as Java and C#. There is also an increasing recognition that Design by Contract (DbC) plays an important role in documenting the contracts between classes [Mey97, BLS02]. In DbC, each class is described by a class invariant as well as preconditions and postconditions for each feature (i.e. method or attribute). DbC is part of the Eiffel language and DbC extensions to Java (e.g. JML) and C# (e.g. Spec#) have now been developed so as to make use of this specification facility for class interfaces [LLM06, BCD^{+}05, BDF^{+}04, BDJ^{+}05, BLS04, BCC^{+}03, FLL^{+}02, JLPS05, LLP^{+}00, LM99, LM06].

Correspondence and offprint requests to: {jonathan, faraz, hhuang}@cse.yorku.ca, bernd.schoeller@inf.ethz.ch. This work was conducted under an NSERC Discovery grant.

The advantage of object-oriented information hiding and contracting mechanisms is that they allow developers to reason statically about the code without the need to consider a vast number of program executions that can occur at run-time. With DbC, developers can reason about code modularly.

Furthermore, contracts are checked dynamically at runtime (via runtime assertion checking) and are therefore useful for testing the correctness of code. The authors of [BLS02] study the use of contracts in practice in order to assess the benefits of doing so for the isolation of faults. Their study shows that instrumenting code with contracts is a powerful lightweight method for detecting bugs and isolating faults.

Concurrency

The situation is somewhat different when it comes to object-oriented concurrent programming. Increasingly, software developers use some variant of concurrency (multithreading, multiprocessing, distributed computing, and web services). But the techniques commonly used to produce concurrent applications are still elementary and often haphazard. The explicit specification and control of low-level parallelism in multithreading models such as Java is a source of new programming errors. Concurrency introduces problems such as data races, atomicity violations and deadlock that make the development and debugging of such software difficult and the resulting products are much more error prone than sequential products. The release of new multi-core CPUs has also stimulated software companies to seek new concurrent languages. Adding more cores to a chip is only half the battle. As stated in [Bie07], the next tricky stage is for software developers to figure out how to program quad cores concurrently so as to avoid "the deadlock bug" and "solve another type of parallel-programming problem called a race bug".

In addition, there is the challenge of extending DbC to the concurrent setting. The standard Hoare rules may fail to apply and more complex calculi are often non-modular making it difficult for use by the developers.

A suitable concurrency model is clearly needed that provides basic safety and liveness guarantees and that shields programmers from common mistakes and errors, or at the very least detects data races, atomicity violations and deadlocks.

One approach is to extend DbC to the concurrent setting as has been done by JML and Spec#. In [RDF$^+$05], JML is extended to allow for the specification of multi-threaded Java programs. Method guards are specified using the **when** keyword. If a feature is called in a state where the guard does not hold, the feature is supposed to block until the guard holds. The new constructs rely on the non-interference notion of method atomicity, and allow developers to specify locking and other non-interference properties of methods. Atomicity enables the specification of method pre- and postconditions and supports Hoare-style modular reasoning about methods. The above keyword is useful in static verification but programmers are still forced to write the synchronization keyword by hand.

As pointed out in [JLPS05], developing safe multithreaded software systems is difficult due to the potential unwanted interference among concurrent threads. In [JLPS05], the Spec# boogie methodology is extended to the concurrent case. The paper presents a method that protects object structures against inconsistency due to race conditions, where developers define aggregate object structures using an ownership system and declare invariants over them. The methodology is supported by a set of language elements and by both a sound modular static verification method and run-time checking support. The method thus ensures freedom from data races and atomicity violations but does not yet treat properties such as freedom from deadlock and other liveness properties.

The JML and Spec# approaches are both based on DbC. Java PathFinder (JPF) [HP00] takes a different approach. JPF takes Java programs (no contracts) as input and provides an impressive verification environment for detecting deadlocks and assertion violations integrating program analysis and model checking. The following quote from [VHB$^+$03] is instructive:

Although it is hard to quantify the exact size of program that JPF can currently handle – "small" programs might have "large" state-spaces – we are routinely analyzing programs in the 1,000 to 5,000 line range ... it is naive to believe that model checking will be capable of analyzing programs of 100,000 lines or more ...

Further progress has been made in the mean time and, undoubtedly, these new methods will be scaled up to handle larger and more realistic examples. Even the ability to analyze small critical chunks of realistic code is a welcome addition to bug detection. Nevertheless, it appears that we will still need to rely on testing for the foreseeable future, with formal verification as a helpful technique for finding additional bugs.

This paper

In this paper, we focus on SCOOP (Simple Concurrent Object-Oriented Programming). SCOOP was originally proposed by Bertrand Meyer [Mey97] with some subsequent attempts to refine or implement the model [Com00, FOP04]. The work of Piotr Nienaltowski [Nie07] is a significant step forward in improving the computational model and providing and a working implementation which includes a library which implements the model (*SCOOPLI*) and a compiler (*scoop2scoopli*) which type checks SCOOP code and translates it into pure threaded Eiffel[1].

SCOOP programs are free of data races and atomicity violations by construction. However, there is no guarantee of freedom from deadlocks. Although the notion of Design by Contract is different to that of sequential Eiffel, contracts in SCOOP permit the application of sequential reasoning techniques to the concurrent programs. In this paper, we explore just how far sequential reasoning takes us and what is needed for verifying arbitrary temporal logic properties of SCOOP programs beyond contracts. The paper proceeds as follows:

- In Section 2 we discuss the difference between sequential Eiffel and SCOOP programs and provide a simple example (class `DATA`) of how theorem proving would work via Hoare reasoning in the sequential case.
- In Section 3 we provide an informal discussion of the SCOOP computational model using a simple example – *zero-one* – to motivate the discussion. The sequential class `DATA` is re-used in this example without the need to make any changes to it for the concurrent case. The Hoare reasoning rule for the concurrent case as described in [Nie07] may be used to verify termination and the contracts of the controlled routines using a theorem prover already in use for sequential Eiffel. The capabilities and the limits of Hoare reasoning are presented. Temporal logic is used to express system properties that are beyond the capabilities of the Hoare rule to verify.
- In Section 4 we provide the outline of a SCOOP Virtual Machine (SVM) together with its operational semantics as a fair transition system. The SVM uses the Hoare rules where applicable to reduce the number of steps in a computation. The SVM can be used as the formal basis for expressing temporal logic properties and for building tools based on modelcheckers and theorem provers to verify safety and liveness properties.

2. Sequential and Concurrent Computation

Object-oriented computation (sequential or concurrent) is performed via the mechanism of the feature call $t.r(x)$. If target t is attached to some object obj then a *processor* may invoke the routine r with argument x on the object obj. In the sequential case, there is only one processor.

In the concurrent case, we have two or more processors. A processor is an autonomous thread of control capable of supporting the sequential execution of instructions for one or more objects. This definition assumes that the processor is some device, which can be implemented either in hardware (e.g. a computer equipped with its own central processor), or as software (e.g. a thread, task or stream). Hence, a processor in this context is an abstraction and we may assume the availability of an unlimited number of processors.

2.1. A Simple Sequential Class

Class `DATA` (Fig. 1) is a simple example of a class written in standard sequential Eiffel. This class will later be re-used to motivate the model for concurrent SCOOP programming. The contracts (preconditions, postconditions and class invariants) document the specification and may also be used to find implementation bugs and to demonstrate the correctness of the code.

Correctness of the implementation can be demonstrated either by run-time *Assertion Testing* or by static

[1] http://se.ethz.ch/research/scoop

compile-time *Formal Verification.* The Hoare correctness rule for a general feature call $t.r(x)$ is as follows:

$$\frac{\{pre_r \wedge I\} do_r \{post_r \wedge I\}}{\{pre'_r\} t.r(x) \{post'_r\}} \quad \text{[SCR – Sequential Correctness Rule]}$$

where pre_r, do_r and $post_r$ are the precondition, body and postcondition of routine r respectively and I is the invariant of the class in which r occurs. The primed notation used in the consequence of rule [SCR] refers to the contracts suitably qualified to the target t. For example, for routine `one` of class `DATA`, rule [SCR] reduces to:

$$\frac{\{x = 0 \wedge y = 0\}\ x := 1;\ y := 1;\ c := c + 1\ \{x = 1 \wedge y = 1 \wedge c = \textbf{old}\ c + 1 \wedge b = \textbf{old}\ b\}}{\{d.x = 0 \wedge d.y = 0\}\ d.one\ \{d.x = 1 \wedge d.y = 1 \wedge d.c = \textbf{old}\ d.c + 1 \wedge d.b = \textbf{old}\ b\}}$$

The job of a theorem prover with respect to class `DATA` is to show the correctness of the antecedent of the rule. Thus, when class `DATA` is re-used in other classes, calls to routines in `DATA` may just use the consequent of the rule. Provided the bodies of the routines in `DATA` have been checked for correctness, clients of `DATA` may then assume that the contracts hold and reason about calls to `DATA` solely on the basis of the contracts without the need to refer to implementation.

Consider, for example, the code in class `TEST` (Fig. 2) which uses class `DATA`. The `create d` instruction (in routine `r`) does a default initialization of all the attributes as shown in the immediately following check statement. Will the feature call `d.one` in the above code succeed without contract violations? All that we need check is that the postcondition of the creation instruction at line 6 entails the precondition of the feature call `d.one` at line 8 (which it does). From the consequent of [SCR] we are guaranteed that the postcondition of routine `one` holds as shown in the check statement at line 9. In modular reasoning, routine `r` in `TEST` is verified only on the basis of its own implementation and the contracts (but not the implementation) of its supplier class `DATA`.

We have implemented such a theorem prover for a significant subset of sequential Eiffel [OWKT06]. This theorem prover trivially verifies the correctness of `DATA` (Fig. 1) and the correctness of the routine `r` in class `TEST` (Fig. 2).

In Assertion Testing, we enable run-time assertion checking and the compiler then generates code that checks the contracts at each feature call (such as `d.one`). Assertion Testing is much weaker than Verification, as [SCR] is only checked for the scenarios written in the testing suite. The benefit is that any code of any size can be automatically checked in this manner without the need to provide complete contracts.

3. The concurrent SCOOP model

The following closely follows the summary of the SCOOP computational model provided in [Nie07, Section 2.4].

Concurrency in SCOOP relies on the basic mechanism of object-oriented computation – the feature call. Each object is handled by a processor referred to as the object's *handler.* All features of a given object are executed by its handler, i.e. only one processor is allowed to access the object. Several objects may have the same handler. The mapping between an object and its handler does not change over time.

If the client and supplier objects are handled by the same processor, a feature call is *synchronous.* If they have different handlers, the call becomes *asynchronous*, i.e. the computation on the client's handler can move ahead without waiting and the call on the object is dispatched to the handler of the supplier object for execution.

An object that is handled by a different processor is called *separate* with respect to the current processor while objects handled by the same processor (as the current processor) are called *non-separate.* A processor, together with the object structure it handles, forms a sequential system. Therefore, every concurrent system may be seen as a collection of interacting sequential systems. Conversely, a sequential system may be seen as a particular case of a concurrent system (with only one processor).

Since each object may be manipulated only by its handler, there is no object sharing between different threads of execution (no shared memory). Given the sequential nature of processors, this results in the absence of intra-object concurrency, i.e. there is never more than one action performed on a given object. Therefore, programs are data-race-free by construction. Locking is used to eliminate *atomicity violations*, i.e.

```
class DATA feature
  x,y,c: INTEGER -- 'c' is a counter
  b: BOOLEAN

  zero is
    require x = 1 and y = 1
    do
        x:=0
        y:=0
        c := c + 1
    ensure
        x = 0 and y = 0 and c = old c + 1 and b = old b
    end

  one is
    require
        x = 0 and y = 0
    do
        x:=1
        y:=1
        c := c + 1
    ensure
        x = 1 and y = 1 and c = old c + 1 and b = old b
    end

  stop is
    do
        b := true
        x := 2
    ensure
        b and x = 2 and y = old y and c = old c
    end

invariant
    ((x = 0 and y = 0) or (y = 1 and x = 1)) or b

end -- class DATA
```

Fig. 1. Class `DATA`

```
class TEST feature
  d: DATA

  r is
    do
      create d
      check d.x = 0 and d.y = 0 and not d.b and d.c = 0 end
      d.one
      check d.x = 1 and d.y = 1 and not d.b and d.c = 1 end
    ensure
      d.x = 1 and d.y = 1 and d.c = 1
    end
end
```

Fig. 2. Class `TEST`

illegal interleaving of calls from different clients. Thus, SCOOP is free of data races and atomicity violations but it is still plagued by the possibility of deadlock.

For a feature call to be valid, it must appear in a context where the client's processor holds a lock on the supplier's processor (this is enforced by type rules). Locking is achieved through feature application – the handler executing a routine with attached formal arguments blocks until the processors handling these arguments have been locked (atomically) for its exclusive use. The routine thus serves, by construction, as the SCOOP version of critical sections. Since a processor may be locked and used by at most one other processor at a time, and all feature calls on a given supplier are executed in a FIFO order, no harmful interleaving occurs.

Condition synchronization relies on preconditions, i.e. a non-satisfied precondition causes waiting. Clients re-synchronize with their suppliers only if and when necessary. Thus, clients wait only on queries (function or attribute calls). This is called *wait by necessity.* Commands (procedure calls) do not require any waiting because they do not yield results that clients would need to wait for.

The language extension supporting the model is quite minimal. SCOOP only needs to enrich Eiffel with type annotations which express the relative locality of objects represented by entities and expressions. An entity may be declared as:

- `x: X`
  Objects attached to `x` are handled by the same processor as the current object. We say that `x` is *non-separate* with respect to **Current**.
- `x:` **separate** `X`
  Objects attached to `x` may be handled by any processor; this processor may (but does not have to) be different from the one handling the current object. We say that `x` is *separate* from **Current**.
- `x:` **separate** `<p> X`
  Objects attached to `x` are handled by a processor known under the name p. The processor tag p may have an unqualified form and be explicitly declared as `p: PROCESSOR`, or have a qualified form derived from the name of another entity, e.g. '`y.handler`'. (`y` must be an attached read-only entity, e.g. a formal argument.) We say that `x` is *separate* from **Current**, and handled by p.

Additionally, a type annotation may include the '`?`' sign, e.g. `x: ? separate X`; this marks a detachable type [ECM06], i.e. the decorated entity may be void (not attached to any object) at run time. Entities not decorated with '`?`' are attached, i.e. not `void`.

[Nie07, Chapter 6] provides a type system to track statically the assignment of objects to processors, i.e. the type system tracks whether an object is local (on the current processor) or separate (on a different processor). The processor tags allow one to express whether several separate objects are on the same processor or not.

In the rest of this section we discuss the semantics of contracts in SCOOP. Of especial interest is the extent to which SCOOP permits the application of sequential reasoning techniques to concurrent programs.

3.1. The zero-one example, schedules and contracts

In standard concurrent programming the naive re-use of library classes that have not been designed for concurrency often leads to data races and atomicity violations. SCOOP allows for the re-use of sequential classes without the need to change the code with synchronization constructs.

We use a very simple example – *zero-one* – to illustrate code re-use, the limits of SCOOP contracts, and to motivate the semantics of the SVM (SCOOP Virtual Machine) which will be developed in the sequel.

In the zero-one example, we re-use class `DATA` (Fig. 1) without the need to add any further synchronization constructs to its code. In addition, our example has classes `ZERO`, `ONE` (Fig. 3) and the root class `ROOT` shown in (Fig. 4).

Execution of the system begins in an initial state in which the root processor π_r has been created and is ready to execute the creation instruction `make` of class `ROOT` (Fig. 4) as shown by the *schedule* s_0 below:

$$[\ \pi_r : \texttt{make}\ ] \tag{1}$$

In the next state, feature call `make` is replaced by the body of the call and we have a new schedule:

$$[\ \pi_r : \texttt{create d; S}\ ] \tag{2}$$

```
class ONE create
   make
 feature
   data: separate DATA
   count: INTEGER

   make(d: separate DATA) is
     do
       data := d
     end

   run is
     do
       from
        until count = 1000
       loop
         do_one(data)
       end
     ensure
        count >= 1000
     end

   do_one(d: separate DATA) is
     require
       not d.b and d.x = 0 and d.y = 0
       count < 1000
     local
       test: BOOLEAN
     do
      d.one
      if d.b then d.stop end -- stop never called
      count := count + 1
      test := d.x = 1
     ensure
       count = old count + 1
       d.x = 1 and d.y = 1 and d.c = old d.c + 1 and not d.b
     end
end -- class ONE
```

Fig. 3. Class `ONE` (similarly for class `ZERO`)

where `S` stands for the remaining statements in the routine `make`. `create d` creates a new processor π_d that handles the instance of the data object just created, and the schedule now looks like:

$$\begin{bmatrix} \pi_r : \mathsf{S} \\ \pi_d : \end{bmatrix} \tag{3}$$

Likewise, an instance of `ONE` is created and handled under the control of processor π_1 and an instance of `ZERO` is created and handled by processor π_0 (lines 11 and 12 of Fig. 4). The various processors with the objects they handle are shown in Fig. 5.

The feature call `launch(zero,one)` at line 13 in Fig. 4 has `Current` as its implicit target. The handler of `Current` is the same processor (π_r) as the enclosing routine `make`, and thus this call is synchronous. Processor π_r must (atomically) reserve both processors π_0 and π_1 (the handlers of the arguments of `launch`) before proceeding to execute the body of `launch`.

The calls `zero.run` and `one.run` (lines 18 and 19 respectively) are asynchronous calls because π_0 and π_r (the handlers of the objects attached to `zero` and `one` respectively) are different processors from the current processor π_r. Thus, these calls must be dispatched to their respective processors (handlers) for execution.

Suppose, instead of wrapping `zero.one` in `launch`, we replace the call `launch` at line 13 with the call `zero.one`. This would be a type error that would be caught at compile-time. Since `zero` is not an argument of the enclosing routine `make`, it cannot be the target of a call in this routine as only those processors associated with arguments are reserved. We do not allow feature calls on targets unless these targets have been reserved (locked) in the current context.

```
class ROOT create
    make
feature
  d: separate DATA
  zero: separate ZERO
  one: separate ONE

  make is
    do
      create d
      create zero.make(d)
      create one.make(d)
      launch (zero, one)
    end

  launch (z: separate ZERO; o: separate ONE) is
    do
      z.run
      o.run
    ensure
      z.count = 1000 and o.count = 1000
    end
end
```

Fig. 4. Class ROOT initiates the three systems

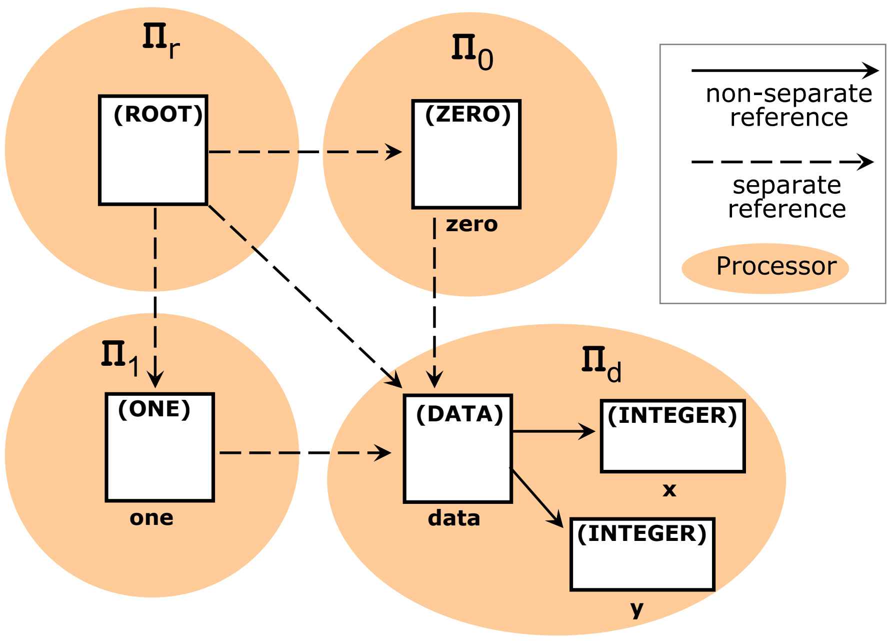

Fig. 5. Processors of the zero-one example

It is instructive to follow an execution of processor π_1 that has arrived at the call `do_one(data)` at line 17 in the context of routine `run` (see Fig. 3). Under SCOOP semantics, the client processor must wait until (a) it gets a lock on the processor (π_d) that handles the argument `data` and (b) the precondition $\neg data.b \wedge d.x = 0 \wedge d.y = 0 \wedge count < 1000$ holds.

At this point modular reasoning breaks down. In the sequential case, it is possible to prove that the body of `run` satisfies its contract (pre- and postconditions) merely by referring to the implementation of the body and the contracts of the features called (such as the contract of `do_one`). However, in the concurrent case, processor π_1 might block forever at line 17 as the processor π_d may never become available. There is thus potential for deadlock at line 17 that cannot be resolved by modular reasoning alone (as the progress of the processor is dependent upon the environment). Thus, some kind of global reasoning will be required.

If the wait conditions (a) and (b) become true, execution continues in the body of `do_one` at line 30 and the schedule is now:

$$\begin{bmatrix} \pi_r : \texttt{S1} \\ \pi_d : \\ \pi_0 : \texttt{S2} \\ \pi_1 : \texttt{data.one; S3} \end{bmatrix} \tag{4}$$

There are no calls pending on processor π_d as the lock has just been obtained. However, execution may proceed on other processors such as π_0 and π_1. For example, if processor π_1 executes next, the feature call `data.one` (line 30 in Fig. 3) will execute asynchronously, i.e. the call must be transferred over to processor π_d for execution. Since π_1 has a lock on π_d the call may be dispatched, immediately, and processor π_1 may continue executing at line 31. The schedule now looks as follows:

$$\begin{bmatrix} \pi_r : \texttt{S1} \\ \pi_d : \texttt{data.one} \\ \pi_0 : \texttt{S2} \\ \pi_1 : \texttt{S3} \end{bmatrix} \tag{5}$$

Processor π_1 waits by necessity at line 31 so that it can check the boolean in the conditional ($d.b$). Execution then continues until line 33 where it does another wait by necessity for the check condition `data.x`. At such waiting points, we wait for all calls to processor π_d to terminate before checking the query. In summary, the SCOOP semantics as proposed in [Nie07] is as follows:

- All preconditions have a *wait semantics*: before executing a routine, the executing handler waits until the precondition is satisfied. A violated precondition results in (possibly infinite) waiting. In practice, the treatment of preconditions is optimized by the run time system to avoid the infinite waiting whenever possible.
- Postconditions keep their usual meaning – they describe the result of a feature application – but each postcondition clause is evaluated individually and asynchronously; a client does not wait unless its handler is involved in a given clause. This increases the amount of concurrency without compromising the guarantees given to the client.
- Loop assertions and check instructions follow a similar pattern as postconditions: they are evaluated by the supplier handler without forcing the client to wait, while still delivering the required guarantees. On the other hand, the individual evaluation of subclauses does not apply; the whole assertion has to hold at the same time even if it involves multiple handlers.
- Class invariants keep their usual semantics because asynchronous calls are prohibited in invariants. This is not imposed by any explicit rule for separate assertions but follows from the refined call validity rule [Nie07].

Every processor has a call stack and a request queue (in our case a row in the schedule). The call stack is used for handling non-separate calls. The request queue stores the feature calls issued by other processors to be handled by this processor. The requests are serviced in the order of their arrival (FIFO).

A processor may be in one of two states: *locked* or *unlocked*. The locked state means that the processor is under the control of another processor and is ready to receive feature requests from that processor alone. The unlocked state means that the processor will not accept any feature calls until it is locked. Each processor repeatedly performs the following actions:

- Request processing: if there is an item on the call stack, pop the item from the stack and process it, i.e.:
  - (Feature application) if the item is a feature request, apply the feature;
  - (Unlocking) if the request queue is empty, set the processor to the unlocked state.
- Request scheduling: if the call stack is empty but the request queue is not empty, dequeue an item and push it onto the stack.
- Idle wait: if both the stack and the queue are empty, wait for new requests to be enqueued.

Before a feature call `t.f(x)` is applied, its formal arguments (suitably bound to actual arguments) must be reserved (i.e. the corresponding processors must be locked) and the feature's precondition must hold. Thus, atomicity of feature calls is enforced and the processor executing feature `f` enjoys exclusive access to to the handlers of the formal arguments of `f`.

3.2. Controlled calls and Hoare reasoning

Consider the following three calls in the zero-one example:

1. Call `one.run` is a separate call that is invoked asynchronously by the root processor π_r (within the context of routine `launch` in class `ROOT`) and sent to processor π_1 for execution. We say that the client π_r *invokes* the call and the supplier π_1 *applies* the feature call.
2. The subsequent call `Current.do_one(data)` (at line 17 in the context of routine `run` in class `ONE`) is a non-separate call invoked synchronously by the current processor π_1.
3. The subsequent call `data.one` (at line 30 in the context of routine `do_one`) is a separate call to processor π_d invoked asynchronously.

The first and third calls are asynchronous and do not wait in the invoking processors as the invoking processor has a lock on the processors that applies the feature calls (at the point calls are invoked). Thus, the calls can be dispatched, immediately, to the invoking processor.

However, as explained earlier, before processor π_1 may invoke `do_one(data)` (the second case above) it must first obtain a lock on the processor π_d handling the argument `data` before it can proceed. What this means is that the enclosing routine `run` in class `ONE` cannot guarantee its own postcondition $count \geq 1000$. This is because there is no guarantee that the lock will be obtained on the processor of the data object. Thus, we cannot reason modularly, i.e., we cannot use the contracts of `do_one(data)` alone to deduce the properties of the enclosing routine `run`. Global (non-modular) reasoning will be needed to verify that the lock will *eventually* be obtained.

A feature call such as `exp1.f(exp2)` involves the evaluation of expressions such as `exp1` and `exp2`. The following definitions will help make the distinction between feature calls in enclosing routines such as `run` (where global reasoning is required) and feature calls in enclosing routines such as `do_one(data)` (where modular reasoning is sufficient). The following definitions are from [Nie07]:

Definition 1 (Controlled Expression). An expression `exp` is *controlled* if and only if it is attached (according to the ECMA definition [ECM06] and either (a) `exp` can be shown to be statically non-separate, or (b) `exp` appears in the context of a routine `r` in which the handler of `exp` is the the same processor as the handler of some attached formal argument of `r`.

Definition 2 (Controlled clause). For a client performing the call `t.f(a)` in the context of a routine `r`, a precondition clause or a postcondition clause of `f` is *controlled* if and only if, after the substitution of the actual arguments `a` for the formal arguments, the clause involves only calls on entities that are controlled in the context of `r`; otherwise, it is uncontrolled.

The check for controlled clauses can be done statically [Nie07]. Consider the call `do_one(data)` at line 17. Its precondition `(not data.b and data.x = 0 and data.y = 0)` is uncontrolled in the context of the enclosing routine `run` as `data` is handled by the separate processor π_d and `data` is not an argument of `run`. In fact, before the call can be invoked processor π_1 must get a lock on the data processor π_d.

By contrast, the precondition `data.x = 0 and data.y = 0` of call `data.one` at line 30 is controlled in the context of routine `do_one` as the calls in the precondition are handled by the same processor π_d as the argument of routine `do_one`. As shown in [Nie07], the Hoare rule for SCOOP is:

$$\frac{\{INV \wedge Pre_r\}\ body_r\ \{INV \wedge Post_r\}}{\{Pre_r^{ctr}[\overline{a}/\overline{f}]\}\ x.r(\overline{a})\ \{Post_r^{ctr}[\overline{a}/\overline{f}]\}} \tag{6}$$

In the rule, there is no distinction between separate and non-separate assertions; both preserve their contractual character. Only controlled assertion clauses are considered by the client; hence the superscript *ctr* decorating them in the conclusion of the rule. From the point of view of the supplier, all assertions occurring in Pre_r and $Post_r$ are controlled; that is why all of them appear in the antecedent.

```
do_one(d: separate DATA) is
  require
    not d.b and d.x = 0 and d.y = 0
    count < 1000
  local
    test: BOOLEAN
  do
   --{precondition holds}
   d.one
   --{not d.b and d.x = 1 and d.y = 1 and d.c = old d.c + 1 and count < 1000}
   if d.b then d.stop
   --{not d.b and d.x = 1 and d.y = 1 and d.c = old d.c + 1 and count < 1000}
   count := count + 1
   --{not d.b and d.x = 1 and d.y = 1 and d.c = old d.c + 1 and count <= 1000 and count = old count + 1}
   test := d.x = 1
   --{test and not d.b and d.x = 1 and d.y = 1 and d.c = old d.c + 1 and count <= 1000 and count = old
        count + 1}
  ensure
    count = old count + 1
    d.x = 1 and d.y = 1 and d.c = old d.c + 1 and not d.b
  end
```

Fig. 6. Hoare reasoning (6) in the concurrent case for controlled routines

Thus, the supplier may assume all the precondition clauses at the the entry to the feature's body and must establish all postcondition clauses when the body terminates. The client must satisfy all the controlled precondition clauses at the moment of the call, but may assume all the controlled postcondition clauses after the call.

All the contracts of feature calls in routine `do_one` are controlled and thus the above Hoare rule may be used to demonstrate the correctness of the implementation with respect to its contracts in the same way as is done in the sequential case as shown in Fig. 6. Thus, a theorem prover can be used to show that routines with controlled contracts are correctly implemented using only the sequential reasoning of the Hoare rule (6).

Since all the contracts of calls in the body of `do_one` are controlled, the Hoare rule provides the client π_1 invoking `d.one` in the body of `do_one` with the same guarantees as there would be in the synchronous case but "projected" into the future. The postcondition of `d.one` is actually evaluated asynchronously by processor π_d in the future after all the calls in the body of routine `d.one` have been executed. However, in the mean time, `do_one` may progress as if the postcondition already holds because the postcondition will indeed hold (eventually) when it becomes relevant at the statement `test := d.x = 1` and again when checking the postcondition of `do_one`.

The same guarantees do not hold for routine `run` in class `ONE` because this routine contains the uncontrolled call `do_one(data)` at line 17.

As defined thus far, the Hoare rule (6) is applicable to feature call `one.run` invoked asynchronously by the root processor π_r (in the context of routine `launch`). The Hoare rule merely says that if routine `run` satisfies its contract then the call `one.run` works according to the rule. However, we know that the current implementation of `run` does not satisfy its postcondition (as discussed above), as the subsequent sub-call `do_one(data)` is not controllable and thus the postcondition cannot be guaranteed (purely on modular reasoning).

For our SVM (SCOOP Virtual Machine) we will need to flag routines such as `launch`. Although all the contracts of clauses in this routine are controlled, the feature calls that implement them are not and hence the contracts cannot be guaranteed modularly via Hoare reasoning. The following definition of a controlled routine can be checked statically:

Definition 3 (Controlled routine). A routine `r(a1, a2, ...)` is controlled if and only if all feature calls in the contract and implementation of the routine are (recursively) controlled.

Following the definition routine `do_one` (in class `ONE`) is controlled but `run` (in class `ONE`) and `launch` (in class `ROOT`) are not.

The implementation of a controlled routine can be verified using standard sequential reasoning. Thus, the sequential Eiffel theorem prover developed in [OWKT06] can be used to verify that the implementation of routine `do_one` satisfies its contracts. Although the routine body can be verified, the call `do_one` in routine `run` is uncontrolled, i.e. we must first wait for the environment to satisfy the precondition before the routine can be executed. However, once the precondition is satisfied, the theorem prover provides a guarantee that the routine will terminate with the postcondition true.

3.3. Detecting deadlocks and liveness properties beyond contracts

The Hoare rule (6) can take us only so far. There are properties beyond modular contractual reasoning that depend upon the emergent behaviour of the complete system. We express these properties below using temporal logic:

1. The safety property $(\pi_r.data.x = 0 = \pi_r.data.y) \vee (\pi_r.data.x = 1 = \pi_r.data.y) \vee b$ can be shown with the Hoare rule. However, the stronger safety property $(\pi_r.data.x = 0 = \pi_r.data.y) \vee (\pi_r.data.x = 1 = \pi_r.data.y)$ cannot.
2. The property $\Box(at_{do\_one} \Rightarrow \Diamond after_{do\_one})$ cannot be demonstrated with the Hoare rule.
3. The property $\Diamond\Box(\pi_r.data.count = 2000)$ cannot be demonstrated with the Hoare rule.

In the next section we outline a SCOOP Virtual Machine which can be used to build tools for checking such properties using a combination of modelcheckers and theorem provers.

4. Outline of a SCOOP Virtual Machine (SVM)

In this section we propose (in broad outline) a SCOOP Virtual Machine (SVM) using the Eiffel memory model described in [Sch06] and the notion of fair transition systems of Manna and Pnueli [MP95] developed in the context of procedural languages for reactive systems. Producing such a virtual machine is challenging as it must integrate:

- The object-oriented memory model (reference semantics, heap, stack etc);
- Concurrent Processing;
- Concurrent Contracts and mechanical verification of these contracts where possible; and
- In the model it must be feasible to verify temporal logic properties.

The Eiffel memory model is based on a small language *Eiffel0* which is a subset of the complete Eiffel language. [Sch06] provides a heap model and operational semantics for the execution of of Eiffel0 programs within the heap. We add to the Eiffel0 model additional features for describing concurrent processors. Eiffel0 does not deal with inheritance, genericity, expanded types and exceptions. Likewise, we only deal with the most significant SCOOP constructs such as creation instructions, preconditions, wait-by-necessity and synchronous and asynchronous routine calls. For brevity, we omit important constructs such as processor tags and runtime checking of postconditions. The intention is to provide an outline of the SVM for the most significant SCOOP constructs.

A *computation* δ of the SVM for a SCOOP program $\mathcal{S}$ is a sequence of *configurations*:

$$\delta = \gamma_0 \longrightarrow \gamma_1 \longrightarrow \gamma_2 \longrightarrow \dots \tag{7}$$

Configurations (denoted γ) will be defined shortly. In the zero-one example of Section 3.1 there were four processors π_r, π_d, π_0 and π_1. The transition from one configuration to another is an atomic step of some processor. Concurrency is represented by the interleaved execution of the atomic instructions of the participating processors as in fair transition systems. Thus, in this approach, concurrency is reduced to nondeterminism where a given concurrent execution gives rise to many possible interleaving orders. Some scheduling constraints will be imposed on computations as the underlying SCOOP scheduler must be fair.

A configuration γ is a tuple $\gamma = (\alpha, \sigma)$ where α is a *schedule* and σ is a *state*. We must now define these notions of schedules and states.

Operations:

$$
\begin{aligned}
\_.\_ &: \mathit{Obj} \times \mathit{AttributeID} \to \mathit{Loc}\\
\_(\_) &: \mathit{Heap} \times \mathit{Loc} \to \mathit{Obj}\\
\_\langle \_ := \_\rangle &: \mathit{Heap} \times \mathit{Loc} \times \mathit{Obj} \to \mathit{Heap}\\
\_^{*} &: \mathit{Heap} \to \mathit{Heap}\\
\mathsf{newproc} &: \mathit{Heap} \to \mathit{Proc}\\
\mathsf{inuse} &: \mathit{Proc} \times \mathit{Heap} \to \mathrm{Bool}\\
\mathsf{new} &: \mathit{Heap} \times \mathit{ClassID} \times \mathit{Proc} \to \mathit{Obj}\\
\_\langle \_\,\mathsf{on}\,\_\rangle &: \mathit{Heap} \times \mathit{ClassID} \times \mathit{Proc} \to \mathit{Heap}\\
\mathsf{alloc} &: \mathit{Obj} \times \mathit{Heap} \to \mathrm{Bool}\\
\mathsf{typeof} &: \mathit{Obj} \to \mathit{ClassID}\\
\mathsf{procof} &: \mathit{Obj} \to \mathit{Proc}\\
\mathsf{lockedby} &: \mathit{Heap} \times \mathit{Proc} \to \mathit{Proc}\\
\_\langle \mathsf{lock}\,\_\,\mathsf{by}\,\_\rangle &: \mathit{Heap} \times \mathit{Proc} \to \mathit{Heap}\\
\mathsf{locked} &: \mathit{Proc} \times \mathit{Heap} \to \mathrm{Bool}
\end{aligned}
$$

Axioms:

$$L \neq K \vee f \neq g \Rightarrow H\langle L.f := X\rangle(K.g) = H(K.g) \tag{H1}$$
$$H\langle X.f := Y\rangle(X.f) = Y \tag{H2}$$
$$H\langle C\,\mathsf{on}\,P\rangle(L) = H(L) \tag{H3}$$
$$H\langle \mathsf{lock}\,P\,\mathsf{by}\,\_\rangle(L) = H(L) \tag{H4}$$
$$\mathsf{alloc}(X, H\langle L := Y\rangle) \Leftrightarrow \mathsf{alloc}(X, H) \tag{H5}$$
$$\mathsf{alloc}(X, H^{*}) \Leftrightarrow \mathsf{alloc}(X, H) \tag{H6}$$
$$\mathsf{alloc}(X, H\langle \mathsf{lock}\,P\,\mathsf{by}\,\_\rangle) \Leftrightarrow \mathsf{alloc}(X, H) \tag{H7}$$
$$\mathsf{alloc}(X, H\langle C\,\mathsf{on}\,P\rangle) \Leftrightarrow \mathsf{alloc}(X, H) \vee X = \mathsf{new}(H, C, P) \tag{H8}$$
$$\mathsf{alloc}(H(L), H) \tag{H9}$$
$$\neg\mathsf{alloc}(\mathsf{new}(H, C, P), H) \tag{H10}$$
$$\mathsf{typeof}(\mathsf{new}(H, C, P)) = C \tag{H11}$$
$$\mathsf{procof}(\mathsf{new}(H, C, P)) = P \tag{H12}$$
$$\mathsf{inuse}(P, H^{*}) \Leftrightarrow \mathsf{inuse}(P, H) \vee P = \mathsf{newproc}(H) \tag{H13}$$
$$\mathsf{inuse}(\mathsf{procof}(H(L)), H) \tag{H14}$$
$$\neg\mathsf{inuse}(\mathsf{newproc}(H), H) \tag{H15}$$
$$\mathsf{lockedby}(H\langle \mathsf{lock}\,P\,\mathsf{by}\,Q\rangle, P) = Q \tag{H16}$$
$$\mathsf{lockedby}(H\langle L := X\rangle, P) = \mathsf{lockedby}(H, P) \tag{H17}$$
$$\mathsf{lockedby}(H\langle C\,\mathsf{on}\,Q\rangle, P) = \mathsf{lockedby}(H, P) \tag{H18}$$
$$P \neq Q \Rightarrow \mathsf{lockedby}(H\langle \mathsf{lock}\,Q\,\mathsf{by}\,R\rangle, P) = \mathsf{lockedby}(H, P) \tag{H19}$$
$$\mathsf{locked}(P, H) = (\mathsf{lockedby}(H, P) \neq \mathsf{const}(\texttt{VoidProc})) \tag{H20}$$

Fig. 7. Abstract Data Type and Axioms for specifying states of the SCOOP Virtual Machine

4.1. Schedules and Configuration transitions

An example of a transition from a pre-configuration to a post-configuration is:

$$\begin{bmatrix} \pi_r : \texttt{S1} \\ \pi_d : \\ \pi_0 : \texttt{S2} \\ \pi_1 : \texttt{data.one};\ \texttt{S3} \end{bmatrix}, \sigma \longrightarrow \begin{bmatrix} \pi_r : \texttt{S1} \\ \pi_d : \texttt{data.one} \\ \pi_0 : \texttt{S2} \\ \pi_1 : \texttt{S3} \end{bmatrix}, \sigma \tag{8}$$

which was discussed in Section 3.1 for the zero-one example. There is no change in state only a reorganization of the queues of the processors involved in the computation. In Section 3.1 the intuitive notion of a schedule was an array of processors where each row of the array is an *action queue*, i.e. a queue of instructions and calls for that processor handled in FIFO order. Since the processor is locked by the calling processor, the

FIFO order guarantees sequential reasoning of the calling processor code free from interference by other processors.

In [Sch06], Instruction is a set of statements of the Eiffel0 language such as assignments, alternatives, loops and calls. We let Statement be the set Instruction together with two additional meta-constructs *popUnlock* and *unlock* which will be defined in Section 4.7. These meta-constructs are used solely in schedule queues at the end of a routine to unlock the processors that the routine has reserved. They are not part of the proper SCOOP language itself.

We let Statement$*$ be a (possibly empty) sequence of statements. In the sequel, *Proc* is an unbounded set of SCOOP processors. Then, a *schedule* α is defined as a map

$$\alpha : Proc \rightarrow \mathsf{Statement}*$$

We use the concrete notation $[\pi_0 : S1,\ \pi_1 : S2, ...]$ as in Section 3.1 for schedules where $\pi_0, \pi_1 \in Proc$ and $S, S1, S2 \in \mathsf{Statement}*$. If we write $[\pi_0 : S1,\ \pi_1 : S2]$ then it is understood that there is an implicit ellipsis as in $[\pi_0 : S1,\ \pi_1 : S2, ...]$. There are an unbounded number of processors but they may not all be in use. Creation instructions on separate targets cause a processer to go to the *in-use* status.

A computation starts in an initial configuration and goes step by step from a configuration (α, σ) to a configuration (α', σ'). We use the "small-step"-like notation:

$$\alpha, \sigma \longrightarrow \alpha', \sigma' \tag{9}$$

to denote a transition from the pre-configuration (α, σ) to the post-configuration (α', σ') as described by the state transition rules in the sequel.

4.2. States and Environments

A *state* σ is a tuple $\sigma = (h, e)$ consisting of a *heap* h and an *environment function* $e : Proc \rightarrow STACK[Env]$. We use the notation σ_h and σ_e to refer to the heap and environment function respectively, i.e. $\sigma = (\sigma_h, \sigma_e)$.

Feature calls are executed with the help of a stack of environments. Through the execution of each routine, this call stack keeps track of arguments and local variables as part of the state. Each processor has its own local stack. Thus we may write the first configuration in (8) as:

$$\begin{bmatrix} \pi_r : \mathtt{S1} \\ \pi_d : \\ \pi_0 : \mathtt{S2} \\ \pi_1 : \mathtt{data.one;\ S3} \end{bmatrix}, \quad (\sigma_h, \begin{bmatrix} \sigma_e^{\pi_r} \\ \sigma_e^{\pi_d} \\ \sigma_e^{\pi_0} \\ \sigma_e^{\pi_1} \end{bmatrix}) \tag{10}$$

where the type of σ_e^{π} is $STACK[Env]$, i.e. σ_e^{π} is the stack for processor π. If $\sigma = (h, e)$ then $e(\pi) = \sigma_e^{\pi}$. The abstract data type for environments is:

$$\begin{aligned} \_(\_) &: Env \times LocalID \rightarrow Obj \\ \_\langle \_ := \_\rangle &: Env \times LocalID \times Obj \rightarrow Env \end{aligned}$$

i.e. an environment is just a store. *Obj* is the set of objects on the heap (defined in Subsection 4.3). *LocalID* is the set of local entities of routines including local variables, routine arguments, `Current` and `Result`. The operations are:

$$E\langle x := y\rangle(x) = y \tag{E2}$$

$$x \neq z \Rightarrow E\langle x := y\rangle(z) = E(z) \tag{E2}$$

Since the type of σ_e^{π} is $STACK[Env]$, all the stack operations apply. Thus $top(\sigma_e^{\pi})$ yields the environment at the top of the stack and $pop(\sigma_e^{\pi})$ yields a new stack the same as the original but with the top environment removed. We may push an environment such as $env = \langle Current := object1, arg1 := object2, ...\rangle$ onto the stack via $push(\sigma_e^{\pi}, env)$ since:

$$push : STACK[Env] \times Env \rightarrow STACK[Env]$$

The environment stores the object attached to `Current` and in the case of a query the object returned by special return variable `Result`.

If we are currently executing in state $\sigma = (h, e)$ on processor π, then we access the current object as $top(\sigma_e^\pi)(\texttt{Current})$. We use the abbreviations σ^π or e^π for the top environment, i.e. $\sigma^\pi = e^\pi = top(\sigma_e^\pi)$. Thus, we can access the object attached to `Current` as $\sigma^\pi(\texttt{Current})$. The notation for environment function override is $e[\pi := push(e(\pi), env)]$, i.e. $e[\pi := push(e(\pi_e), env)]$ is an environment function the same as e except that the call stack for processor π is $push(e(\pi), env)$ which is the same as the old call stack with the additional environment env at the top.

4.3. The Heap

The precise effect of SCOOP instructions (e.g. creation instructions, assignments and feature calls) on the state will be defined in detail below. For that we will need abstract data types for expressing updates to the heap as shown in Fig. 7. The first three $Heap$ operations in Fig. 7 are:

$$\begin{aligned} \_.\_ &: Obj \times AttributeID \to Loc \\ \_(\_) &: Heap \times Loc \to Obj \\ \_\langle \_ := \_\rangle &: Heap \times Loc \times Obj \to Heap \end{aligned}$$

Obj (set of objects), Loc (the set of locations, or fields of an object), and $Proc$ (the set of concurrent processors) are disjoint sets. Given an object $o \in Obj$ (say of type $DATA$ with integer attribute "x"), then we may use the first operation (e.g. $o.x$) to access the location (i.e. the field in o associated with attribute x). The second operation allows us to obtain the object associated with a field in a heap (e.g. $h(o.x) = o$).

The third operation allows us to update a field (e.g $o.x$) in a heap (e.g. h). Thus, $h\langle o.x := 3\rangle$ is a heap the same as h, except with the field $o.x$ set to integer constant 3. We can thus use the notation repetitively to get a set of simultaneous updates, e.g. $h\langle o.x := 1\rangle\langle o.y := 1\rangle$ is the heap the same as heap h except where the fields associated with x and y in the data object have both been updated to the integer constant 1.

4.4. Creation instructions

Consider the following operations of the memory model are:

$$\begin{aligned} \_^* &: Heap \to Heap \\ \mathsf{newproc} &: Heap \to Proc \\ \mathsf{inuse} &: Proc \times Heap \to \mathrm{Bool} \\ \mathsf{new} &: Heap \times ClassID \times Proc \to Obj \\ \_\langle \_ \,\mathsf{on}\, \_\rangle &: Heap \times ClassID \times Proc \to Heap \\ \mathsf{alloc} &: Obj \times Heap \to \mathrm{Bool} \\ \mathsf{typeof} &: Obj \to ClassID \\ \mathsf{procof} &: Obj \to Proc \end{aligned}$$

In order to understand how these operations are used, consider the code snippet:

```
class ROOT create
    make
feature
  d: separate DATA
  ...
  create d
  ...
end
```

Informally, the separate creation semantics is:

1. Create a new processor (say π_d).
2. Create a new object o_d of type $DATA$ that runs on processor π_d.
3. Attach the field of the object associated with the entity "d" to the new object o_d.

Consider the separate creation instruction:

```
d: separate DATA
...
create d
```

$$\frac{}{[\pi_r : \textbf{create}\ \text{d};\ ...]\,,\ \sigma\ \longrightarrow\ \begin{bmatrix} \pi_r : ... \\ \pi_d : \end{bmatrix},\ (\sigma'_h,\ \sigma_e)}\quad [\text{ T1 – separate create }]$$

where
$\sigma'_h = \sigma^*_h\langle \mathit{DATA}\ \textbf{on}\ \pi_d\rangle\langle\sigma^{\pi_r}(\mathit{Current}).d := o_d\rangle$
$o_d = \mathit{new}(\sigma_h, \mathit{DATA}, \pi_d)$
$\pi_d = \mathit{newproc}(\sigma_h)$

Fig. 8. Rule T1: separate create

For the non-separate creation instruction:

```
a: CA
...
create a
```

$$\frac{}{\begin{bmatrix} \pi_r : \textbf{create}\ \text{a}\ ;\ ... \\ ... \end{bmatrix},\ \sigma\ \longrightarrow\ \begin{bmatrix} \pi_r :\ \ ... \\ ... \end{bmatrix},\ (\sigma'_h,\ \sigma_e)}\quad [\text{ T2 – non-separate create }]$$

where
$\sigma'_h = \sigma_h\langle \mathit{CA}\ \textbf{on}\ \pi_r\rangle\langle\sigma^{\pi_r}(\mathit{Current}).a := o_a\rangle$
$o_a = \mathit{new}(\sigma_h, \mathit{CA}, \pi_r)$

Fig. 9. Rule T2: non-separate create

Rule [T1] uses the memory model to specify the precise relationship of the post-state σ to the pre-state σ' for the separate creation instruction at line 6.

The antecedent of rule [T1] is empty meaning that the state transition can always be taken from the configuration associated with the pre-state. Other rules in the sequel will have *enabling conditions* which state under what conditions the transition occurs. These enabling conditions will be stated above the line in the rule.

Creation instructions may also have a creation procedure. In such a case, the application processor is locked, the creation routine is asynchronously dispatched to the application processor for execution as in rule [T4] in the sequel (allowing the invoking processor to make progress), and the lock is then released. The attribute d is attached to the newly created object as soon as the object is created but without waiting for the creation procedure to terminate. As a result, d is now pointing to an inconsistent object. However, this is not harmful because the object cannot be accessed before its handler is released. This in turn cannot happen until the creation routine terminates and the processor is unlocked, at which point the object is in a consistent state.

An entity `a` of type `CA` which is declared as non-separate has the simpler creation rule [T2]. In this rule there is no need to create a new processor. The new instance of class `CA` is handled by the current processor.

4.5. Expressions

Following [Sch06] we describe below the operation Expr that is applied to a SCOOP expression to compute its value. The value of an expressions is always an object in *Obj*. As in [Sch06], expressions are assumed to be side-effect free following the command-query separation rule. Thus commands can change the state but not queries.

Expressions such as constants, attributes and query calls are used in contracts and program instructions.

For example, in the assignment `z := t.q(x)` the expression `t.q(x)` is a query routine that returns an object o and the heap must be updated so that attribute `z` now points to o. For simplicity we allow only those query routines where all calls in the body of the query are handled by a single processor (as is the case in the zero-one example). Thus (unlike command routines) we do not allow a query routine handled by some processor to make a call to another processor. If a query is handled by the current processor then it may be evaluated immediately (as all relevant commands that determine the value of the query will already have been executed). If the query is handled by a separate processor, then the current processor must wait until all the commands sent to that processor's action queue have been executed (and the action queue is empty) before the query can be evaluated. As mentioned earlier – this is called "wait by necessity" – and its rule will be provided in the sequel.

To retain a pure object-oriented logic based on objects and references, we model the basic value types such as booleans and integers through immutable objects. Constant identifiers (of the set $ConstID$) describe these objects. The set contains identifiers such as `true`, `false`, `Void` and `VoidProc`. The related immutable object can be looked up through the function $\mathsf{const} : ConstID \rightarrow Obj$.

The evaluation function Expr in [Sch06] of an expression returns an object in Obj that corresponds to the evaluation of the expression in a state (where the state is a tuple consisting of a heap and an environment) using a terminating "big-step"-like semantics. Since queries in this paper are handled atomically by some processor (perhaps different from the current processor) and are side-effect free we may re-use the semantics in [Sch06] by adding to the state the processor that evaluates the expression. In [Sch06], an expression E is evaluated in a state consisting of the tuple $(heap, environment)$ whereas, for us, expression E is evaluated in the tuple (π, σ) where π is the processor and σ is the state as defined in Section 4.2.

We cannot re-use the semantics of [Sch06] for commands. This is because commands change the state and must be executed in order on the correct processors as will be described in transition rules [T4] – [T6].

In the definitions below, $c \in ConstID$. In the query $q(arg1, \dots)$ let $l, arg1, arg2 \in LocalID$ (i.e. these are local entities of the query including local variables, arguments, `Current` and `Result`).

$$\mathsf{Expr}[\![c]\!](\pi,\sigma) = \mathsf{const}(c) \tag{11}$$

$$\mathsf{Expr}[\![l]\!](\pi,\sigma) = \sigma^{\pi}(l) \tag{12}$$

$$\mathsf{Expr}[\![\texttt{Current}]\!](\pi,\sigma) = \sigma^{\pi}(\texttt{Current}) \tag{13}$$

$$\mathsf{Expr}[\![a]\!](\pi,\sigma) = \sigma_h(\sigma^{\pi}(\texttt{Current}).a) \tag{14}$$

$$\mathsf{Expr}[\![E.a]\!](\pi,\sigma) = \sigma_h(\mathsf{Expr}[\![E]\!](\pi,\sigma).a) \tag{15}$$

$$\mathsf{Expr}[\![E1 = E2]\!](\pi,\sigma) = (\mathsf{Expr}[\![E1]\!](\pi,\sigma) = \mathsf{Expr}[\![E2]\!](\pi,\sigma)) \tag{16}$$

To evaluate a side-effect free query routine, the rule is:

$$\frac{\begin{array}{l} env = \langle Current := \mathsf{Expr}[\![T]\!](\pi,\sigma),\ arg_1 := \mathsf{Expr}[\![A1]\!](\pi,\sigma), \dots\rangle \\ Body_q,\ (\sigma_h, env) \quad \longrightarrow \quad (h', env') \end{array}}{\mathsf{Expr}[\![T.q(A1, A2, \dots)]\!](\pi,\sigma) = env'(\texttt{Result})}$$

The above rule is in the format of [Sch06] and can thus use the sequential machinery of [Sch06] to evaluate the value returned by the query.

In postconditions (**ensure** clauses), **old** references the pre-state of the computation. Thus, a postcondition is a double state formula $Post$, i.e. it is relation between the pre-state and the post-state. As in the case of single state formulas we can define an operation $\mathsf{OExpr}[\![Post]\!](\pi,\sigma,\sigma')$ by analogy with the same operation in [Sch06].

The transition rules for the standard Eiffel0 commands of SCOOP follow [Sch06] adapted for configuration transitions as in this paper. For example, for the alternative `if b then S1 else S2` we have two rules one for the case where the boolean guard `b` evaluates to true and one for where it evaluates to false. In the former case, the rule is:

$$\frac{\mathsf{Expr}[\![b]\!](\pi,\sigma) = \mathsf{const}(\texttt{true})}{\left[\begin{array}{l} \pi : \texttt{if b then S1 else S2; S3} \\ \dots \end{array}\right], \sigma \quad \longrightarrow \quad \left[\begin{array}{l} \pi : \texttt{S1; S3} \\ \dots \end{array}\right], \sigma} \quad [\text{ Alternate} - \mathsf{Expr}[\![b]\!] \text{ holds }]$$

Consider the code:

```
z: separate Z
...
r(t: separate T; x: separate X)
  do
      t.command1
      t.command2
      z := t.q(x) −− wait by necessity
  end
```

The inference rule for the wait by necessity is:

$$\frac{\begin{array}{l}(1)\ \pi_a \neq \pi_b \\ (2)\ \sigma = (h, e) \\ (3)\ \mathit{procof}(o) = \pi_b \text{ where } o = \mathsf{Expr}[\![t]\!](\pi_a, \sigma) \\ (4)\ \mathit{env} = \langle \mathtt{Current} := o, \mathtt{arg} := \mathsf{Expr}[\![x]\!](\pi_a, \sigma)\rangle \\ (5)\ e' = e[\,\pi_b := \mathit{push}(e(\pi_b), \mathit{env})\,] \\ (6)\ \mathsf{Expr}[\![q.\mathit{pre}]\!](\pi_b, (h, e')) = \mathsf{const}(\mathtt{true}) \\ (7)\ h'' = h\langle \mathsf{Expr}[\![\mathtt{Current}]\!](\pi_a, \sigma).z := \mathsf{Expr}[\![t.q(x)]\!](\pi_a, \sigma)\rangle\end{array}}{\begin{bmatrix}\pi_a : \mathtt{z := t.q(x);\ S1} \\ \pi_b :\end{bmatrix}, (h, e) \longrightarrow \begin{bmatrix}\pi_a : \mathtt{S1} \\ \pi_b :\end{bmatrix}, (h'', e)} \quad [\text{ T3 – Wait by Necessity }]$$

Fig. 10. Rule T3: Wait by Necessity

$$\frac{\begin{array}{l}\pi_a \neq \pi_b \\ \mathit{procof}(o) = \pi_b \text{ where } o = \mathsf{Expr}[\![t]\!](\pi_a, \sigma)\end{array}}{\begin{bmatrix}\pi_a : \mathtt{t.f(a);\ S1} \\ \pi_b : \mathtt{S2}\end{bmatrix}, \sigma \longrightarrow \begin{bmatrix}\pi_a : \mathtt{S1} \\ \pi_b : \mathtt{S2;\ t.f(a)}\end{bmatrix}, \sigma} \quad [\text{ T4 – Asynch Call }]$$

Fig. 11. Rule T4: Asynch Call

4.6. Wait by Necessity

Transition Rule [T3] (Fig. 10) shows the move from a pre-configuration to a post-configuration for an assignment on processor π_a given by `z := t.q(x)` where the processer that handles the query routine q is π_b as shown in the enabling condition (3).

The transition may only be taken when the the action queue of π_b is empty (i.e. has already processed the commands `command1` and `command2`) and the precondition of query routine q must be true (6). The pre-configuration schedule thus indicates an empty action queue for processor π_b. The precondition may only be evaluated after the formal argument of the query `arg` is bound to the actual argument `x` as described in (4) and (5). Finally, once the transition is taken, `z` is attached to the value calculated by the query which is the heap update shown in (7).

4.7. Command routine calls – controlled and uncontrolled

In the zero-one example of Section 3.1, calls such as `one.run` (in routine `launch` of class `ROOT`) and `d.one` (in routine `do_one` of class `ONE`) are invoked asynchronously. In each case, the invoking processor already has a lock on the application processor and thus the feature call can be transferred over, immediately, from the invoking processor to the application processor. The *Asynch Call* Rule [T4] provides a description of such state configuration transitions where a call `t.f(a)` is transferred from a processor π_a to a processor π_b provided the target `t` is handled by processor π_b (this is the enabling condition of this state transition).

The compile-time check guarantees the Call Validity Rule [Nie07, Rule 6.5.3], i.e. the target of a feature call must be controlled (the target is controlled if and only if it is attached and non-separate or has the same processor as some attached formal argument of the enclosing routine). The attachability requirement eliminates calls on void targets. The additional conditions ensure the correct locking of the target – the target

[T5] – Uncontrolled Synchronous Call

Uncontrolled synchronous routine invocation `t.r(x1,x2)` on some processor π_a with separate arguments `x1` and `x2` not yet reserved. The formal arguments of routine `r` are `arg1` and `arg2` for actual arguments `x1` and `x2` respectively.

$$\frac{\begin{array}{l}(1)\ \pi_a \neq \pi_b \wedge \pi_a \neq \pi_c \\ (2)\ \sigma = (h, e) \\ (3)\ \mathsf{procof}(o) = \pi_a \text{ with } o = \mathsf{Expr}[\![t]\!](\pi_a, \sigma) \\ (4)\ \mathsf{procof}(\mathsf{Expr}[\![x1]\!](\pi_a, \sigma)) = \pi_b \wedge \mathsf{procof}(\mathsf{Expr}[\![x2]\!](\pi_a, \sigma)) = \pi_c \\ (5)\ \neg\mathsf{locked}(\pi_b, \sigma_h) \wedge \neg\mathsf{locked}(\pi_c, \sigma_h) \\ (6)\ \mathsf{Expr}[\![r.pre]\!](\pi_a, (h', e')) = \mathsf{const}(\mathtt{true}) \\ (7)\ h' = h < \mathsf{lock}\ \pi_b\ \mathsf{by}\ \pi_{\mathsf{a}} >< \mathsf{lock}\ \pi_c\ \mathsf{by}\ \pi_{\mathsf{a}} > \\ (8)\ e' = e[\pi_a := push(e(\pi_a), < \mathtt{Current} := o, \mathtt{arg1} := \mathsf{Expr}[\![x1]\!](\pi_a, \sigma), \mathtt{arg2} := \mathsf{Expr}[\![x2]\!](\pi_a, \sigma) >)\end{array}}{\left[\begin{array}{l}\pi_a : \mathtt{t.r(x1,x2);\ S1} \\ \pi_b : \\ \pi_c :\end{array}\right], (h, e) \longrightarrow \left[\begin{array}{l}\pi_a : \mathsf{body}_r;\ popunlock(\pi_b, \pi_c);\ \mathtt{S1} \\ \pi_b : \\ \pi_c :\end{array}\right], (h', e')}\ [\ T5a\ ]$$

Rule [T5b] for $popunlock(\pi_b, \pi_c)$:

$$\frac{}{\left[\begin{array}{l}\pi_a : popunlock(\pi_b, \pi_c);\ \mathtt{S1} \\ \pi_b : \mathtt{S2} \\ \pi_c : \mathtt{S3}\end{array}\right], (h, e) \longrightarrow \left[\begin{array}{l}\pi_a : \mathtt{S1} \\ \pi_b : \mathtt{S2};\ unlock \\ \pi_c : \mathtt{S3};\ unlock\end{array}\right], (h,\ e[\pi_a := pop(e(\pi_a))])}\ [\ T5b\ ]$$

Rule [T5c] for *unlock*:

$$\frac{}{\left[\begin{array}{l}\pi : unlock \\ \ldots\end{array}\right], (h, e) \longrightarrow \left[\begin{array}{l}\pi : \\ \ldots\end{array}\right], (h < \mathsf{lock}\ \pi\ \mathsf{by}\ \mathtt{VoidProc} >,\ e)}\ [\ T5c\ ]$$

Fig. 12. Rules T5a, T5b and T5c: Uncontrolled Synch Call

must be handled either synchronously by the current processor or asynchronously by the an application processor handling one of the locked arguments, in which case the current processor has a lock on the call application processor. Thus, the Call Validity Rule ensures that whenever we have a pre-schedule as the one shown in Rule [T4], the handler π_b of the entity `t` will be locked by π_a. Rule [T4] enforces the FIFO nature of the call queue of π_b because the call `t.f(a)` is placed at the end of the call queue of π_b as shown in the post schedule.

The uncontrolled synchronous rule [T5a] applies to feature calls such as `launch(zero,one)` in class `ROOT`. The call is initiated only if the root processor π_r is able to obtain locks on the processors of `zero` and `one`, i.e. they are currently unlocked (5) and the precondition of routine `launch` holds (6). Under these conditions the body of `launch` can be executed provided the actual arguments are substituted for the formal arguments. [T5a] replaces the feature call with its body. When the body terminates (reaches its **end** statement) the calling processor unlocks the application processors as shown in [T5b] and [T5c].

Rule [T5] also applies to the routine call `do_one(data)` in class `ONE`. However, in this case we can obtain a significant reduction in the number of steps that must be executed using Rule [T6]. Since all the feature calls in the contracts and implementation of `do_one(data)` are controlled, we can demonstrate the correctness of the implementation of this routine with respect to its contracts using a theorem prover (using Hoare reasoning (6) as shown in Fig. 6). Thus, once the locks are obtained on the processors of the formal arguments (in this case `data`) and the precondition holds, the SVM can move to a new configuration that satisfies the postcondition of the routine (without the need to execute the body of the routine).

Rule [T6] results in fewer transition steps than [T5] due to the fact that the many steps in the body is replaced by single step involving the contractual relation $\rho(\sigma, \sigma')$ between the pre-state and the post-state. The downside is that the computation now becomes symbolic because the contracts do fully specify the changes to the state. This problem can be ameliorated by the use of dynamic frames suggested in [SO06].

Rule for a synchronous routine call `t.r(x1,x2)` where the routine itself is controlled i.e. all feature calls in its contracts and implementation are themselves controlled as in Definition 3 and the implementation satisfies the contract. The routine has a precondition *r.pre* and a postcondition *r.post*.

$$\frac{\begin{array}{l}
(1)\ \pi_a \neq \pi_b \wedge \pi_a \neq \pi_c \\
(2)\ \mathsf{procof}(o) = \pi_a \text{ with } o = \mathsf{Expr}[\![t]\!](\pi_a, \sigma) \\
(3)\ \mathsf{procof}(\mathsf{Expr}[\![x1]\!](\pi_a, \sigma)) = \pi_b \wedge \mathsf{procof}(\mathsf{Expr}[\![x2]\!](\pi_a, \sigma)) = \pi_c \\
(4)\ \neg\mathsf{locked}(\pi_b, \sigma_h) \wedge \neg\mathsf{locked}(\pi_c, \sigma_h) \\
(5)\ \mathsf{Expr}[\![r.pre]\!](\pi_a,\ (\sigma_h,\ \sigma_e[\pi_a := push(\sigma_e(\pi_a), env)])\,) = \mathsf{const}(\mathtt{true}) \\
\text{where } env = \langle \mathtt{Current} := 0, \mathtt{arg1} := \mathsf{Expr}[\![x1]\!](\pi_a, \sigma), \mathtt{arg2} := \mathsf{Expr}[\![x2]\!](\pi_a, \sigma) \rangle \\
(6)\ \rho(\sigma, \sigma') \\
\text{where } \rho(\sigma, \sigma') \overset{\text{def}}{=} \quad \mathsf{Expr}[\![r.pre]\!](\pi_a, \sigma) = \mathsf{const}(\mathtt{true}) \wedge \mathsf{Expr}[\![INV]\!](\pi_a, \sigma) = \mathsf{const}(\mathtt{true}) \wedge \\
\quad\quad \mathsf{OExpr}[\![r.post]\!](\pi_a, \sigma, \sigma') = \mathsf{const}(\mathtt{true}) \wedge \mathsf{Expr}[\![INV]\!](\pi_a, \sigma') = \mathsf{const}(\mathtt{true})
\end{array}}{\left[\begin{array}{l} \pi_a : \mathtt{t.r(x1,x2);\ S1} \\ \pi_b : \\ \pi_c : \end{array}\right], \quad \sigma \quad \longrightarrow \left[\begin{array}{l} \pi_a : \mathtt{S1} \\ \pi_b : \\ \pi_c : \end{array}\right], \quad \sigma'} \quad [\text{ T6 – Controlled Routine }]$$

Note that $\rho(\sigma, \sigma')$ holds provided the implementation of `r` satisfies its contract (e.g. as shown by a theorem prover).

Fig. 13. Rule T6: Controlled Routine

4.8. Fair transition systems and Temporal Logic

The SVM executes the configuration transitions defined in the previous subsections, i.e. a computation δ of the SVM is given by

$$\delta = \gamma_0 \longrightarrow \gamma_1 \longrightarrow \gamma_2 \longrightarrow \ldots \tag{17}$$

where each change in configuration is an action taken by one of the processors of the SVM.

We require that the scheduling be fair. Rules [T5] and [T6] have enabling conditions that depend upon getting the lock on multiple processors and thus will require a strong fairness constraint called *compassion* on the computations [MP95]. In compassion, it is not the case that the transition is enabled infinitely many times but not taken beyond some point. Thus, in these rules, if the associated configuration transitions continually become simultaneously available (perhaps interspersed with periods of unavailability) they must eventually be taken. The implementation of SCOOP in [Nie07] via a global scheduler guarantees the fairness of the transitions described in [T5] and [T6] of the SVM.

For the rest of the rules we require *justice*, i.e. it is not the case that the transition associated with the rules is continuously enabled beyond some point and yet not taken.

For a SCOOP program S, we let Δ_S be the set of all computations generated by the SVM via the rules with the fairness constraints. As mentioned at the beginning of this section the transitions of the processes are interleaved in a given computation and thus concurrency is modeled by non-determinism. The fairness constraints remove from Δ_S those computations that do not meet these constraints and thus Δ_S distinguishes concurrency from pure nondeterminism [MP92, p129].

In Section 3.3 we showed that certain temporal logic properties such as $\Diamond\Box(\pi_r.data.count = 2000)$ are beyond the proof capabilities of the Hoare rule (6). The SVM provides us with the necessary machinery for defining when a SCOOP program satisfies a temporal logic property. We can interpret a temporal logic formula ψ in a computation δ in the standard way (notation: $\delta \vDash \psi$). Thus, given a SCOOP program S and a temporal logic formula ψ

$$(\text{SCOOP program } S \text{ satisfies temporal logic formula } \psi) \quad \text{iff} \quad (\forall\, \delta \in \Delta_S \bullet \delta \vDash \psi) \tag{18}$$

We have thus provided an outline of the SVM that can be used to check the correctness of temporal logic properties. To check system properties beyond the ones that the theorem prover for contracts can handle, we could rely on model checking and theorem proving techniques for fair transition systems. For example, we could envisage using the SPIN tool [Hol97] or other such efficient state exploration tools to implement the SVM.

5. Conclusion and future work

SCOOP eliminates race conditions and atomicity violations by construction. However, it is still vulnerable to deadlocks.

In this paper we described how far the SCOOP notion of contracts can take us in verifying interesting properties of concurrent system using modular Hoare rules. The Hoare rule can be used to verify termination and the contracts in the case that the routine is controlled. Techniques developed for sequential Eiffel can be applied as is to these routines (Section 3.2).

Some safety and liveness properties depend upon the environment and cannot be proved using the Hoare rules. To deal with such system properties, we described, in outline, a SCOOP Virtual Machine (SVM) as a fair transition system (Section 4). The SVM makes it feasible to use modelchecking and theorem proving methods for checking global temporal logic properties of SCOOP programs. The SVM uses the Hoare rules where applicable to reduce the number of steps in a computation.

We are currently exploring the SPIN model checker [Hol97] for implementing the SVM. We have been able to verify small programs. However, we have not yet used symbolic execution as proposed in rule [T6] to handle larger and more realistic cases.

References

[BCC+03] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. In *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, 2003.

[BCD+05] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Proceedings of FMCO*, 2005.

[BDF+04] Mike Barnett, Robert DeLine, Manuel Fahndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[BDJ+05] Mike Barnett, Robert DeLine, Bart Jacobs, Manuel Fhndrich, K. Rustan M. Leino, Wolfram Schulte, and Herman Venter. The Spec# Programming System: Challenges and Directions. *Position paper at VSTTE*, 2005.

[Bie07] Celeste Biever. Chip revolution poses problems for programmers. *NewScientist*, 193(2594):26–27, 2007.

[BLS02] L. C. Briand, Y. Labiche, and H. Sun. Investigating the use of analysis contracts to support fault isolation in object oriented code. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 70–80, New York, NY, USA, 2002. ACM Press.

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004*. Springer Verlag, LNCS 3362, 2004.

[Com00] M. Compton. SCOOP: an Investigation of Concurrency in Eiffel. Master's thesis, Department of Computer Science, The Australian National University, 2000.

[ECM06] ECMA. Eiffel: Analysis, design and programming language. Standard ECMA-367 (2nd edition), June 2006.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.

[FOP04] Oleksandr Fuks, Jonathan S. Ostroff, and Richard F. Paige. SECG: The SCOOP-to-Eiffel Code Generator. *JOT Journal of Object Technology*, 11(3), 2004.

[Hol97] Gerard Holzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.

[HP00] K. Havelund and T. Pressburger. Model checking Java programs using Java pathfinder. *Software Tools for Technology Transfer (STTT)*, 2(4):72–84, 2000.

[JLPS05] Bart Jacobs, K. Rustan M. Leino, Frank Piessens, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, pages 137–146. IEEE, 2005.

[LLM06] Gary T. Leavens, K. Rustan M. Leino, and Peter Muller. Specification and verification challenges for sequential object-oriented programs. TR 06-14, Department of Computer Science, Iowa State University, May 2006.

[LLP+00] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106, 2000.

[LM99] K. Rustan M. Leino and Rajit Manohar. Joining Specification Statements. *Theoretical Computer Science*, 216(1–2):375–394, 1999.

[LM06] K. Rustan M. Leino and Peter Mller. A verification methodology for model fields. In *ESOP'06*, 2006.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, New York, 1992.

[MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[Nie07] Piotr Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming.* PhD thesis, ETH ZURICH, 2007.

[OWKT06] Jonathan Ostroff, Chen-Wei Wang, Eric Kerfoot, and Faraz Ahmadi Torshizi. Automated model-based verification of object oriented code. In *Verified Theories: Theories, Tools, Experiments (VSTTE Workshop, Floc 2006)*. Microsoft Research MSR-TR-2006-117, 2006.

[RDF+05] Edwin Rodríguez, Matthew B. Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. In *ECOOP*, pages 551–576, 2005.

[Sch06] Bernd Schoeller. Eiffel0: An object-oriented language with dynamic frame contracts. Technical Report 542, ETH Zurich, 2006.

[SO06] Bernd Schoeller and Jonathan Ostroff. Dynamic frame contracts. *Submitted for publication*, 2006.

[VHB+03] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering Journal*, 10, 2003.