Contracts for concurrency

Piotr Nienaltowski, Bertrand Meyer

Chair of Software Engineering
Swiss Federal Institute of Technology (ETH)
8092 Zurich, Switzerland
{Piotr.Nienaltowski, Bertrand.Meyer}@inf.ethz.ch

Abstract. The SCOOP model extends the Eiffel programming language to provide support for concurrent programming. The model is largely based on the principles of Design by Contract. Nevertheless, the semantics of contracts used in SCOOP is not suitable for concurrent programming because it only allows for restricted reasoning about correctness properties; liveness properties are completely intractable. Additionally, SCOOP does not provide a clear semantics for postconditions. We propose a generalized semantics of preconditions, postconditions, and invariants that is applicable in concurrent and sequential contexts. We demonstrate how this semantics may be used for reasoning about correctness of SCOOP programs. We also analyze the relation between assertion violations and deadlocks. We illustrate the discussion with several examples.

1 Introduction

Design by Contract [1] allows programmers to equip class interfaces with *contracts*. Through the use of assertions, contracts express the mutual obligations of *clients* and *suppliers*. Routine *preconditions* specify the obligations on the routine client and the guarantee given to the routine supplier. Conversely, routine *postconditions* express the obligation on the routine supplier and the guarantee given to the routine client. Class *invariants* express the correctness criteria of a given class — an instance of a class is in a consistent state if and only if the corresponding invariant holds in every observable state.

The modular design fostered by Design by Contract reduces the complexity of software — correctness considerations can be confined to the boundaries of components (classes) that can be proved and tested separately. Clients can rely on the interface of a supplier without the need to know about its implementation details. We define the correctness of a class as follows.

Definition 1. Local correctness (sequential). *Routine r of class C is locally correct iff after the execution of r's body, both the class invariant Inv_C and the postcondition $Post_r$ of that routine hold, provided that both the invariant and the precondition Pre_r were fulfilled at the time of the invocation.*

Following the principles of Design by Contract, it is possible to reason about the correctness of feature calls using a simple rule:

$$\frac{\{INV \wedge Pre_r\}\ \ body_r\ \ \{INV \wedge Post_r\}}{\{Pre_r[\overline{a}/\overline{x}]\}\ \ r(\overline{a})\ \ \{Post_r[\overline{a}/\overline{x}]\}} \tag{1.1}$$

Rule 1.1 states that if feature *r* is locally correct then a call to that feature executed in a state that satisfies its precondition will terminate in a state that satisfies its postcondition (with actual arguments substituted for formal arguments). This is very convenient for proving correctness of sequential programs – clients have to ensure that the precondition holds before the call and they may assume the postcondition after the call.

It is tempting to apply the same rule to reasoning about SCOOP programs [2]. Unfortunately, the assertion mechanism based on the standard semantics for preconditions and postconditions breaks down in concurrent setting. Consider feature *store* in figure 1. Its precondition states that *a_buffer* must not be full when the feature is called. So, in order to prove that the call to *store* (*buffer*) appearing in feature *produce* is correct, it is necessary to show that **not** *buffer.is_full* holds at the moment of the call. But the client has no possibility to ensure that it holds — since *buffer* denotes a separate object, other clients may modify its state and invalidate the precondition in the meantime. This problem is known as *concurrent precondition paradox* — suppliers cannot do their work without the guarantee that the precondition holds; but for separate arguments the clients are unable to ensure these preconditions. To solve the problem, Meyer [3] proposes a new semantics for precondition clauses involving separate calls — such preconditions become *wait-conditions*. They make the caller of the routine wait until all wait-conditions are satisfied. To be more precise, a call to *store* (*buffer*) will block until (1) the processor that handles the object represented by *buffer* is reserved for the exclusive use of the client and (2) wait-condition **not** *buffer.is_full* holds. But the rule does not require the client to ensure these two conditions. As a result, we cannot decide whether the call to *store* (*buffer*) will ever proceed! So, the problem is palliated but not solved. To solve the problem completely, we need a rule that takes into account potential interference of several processors present in a SCOOP system.

Note that, in SCOOP, wait-conditions appear as part of a routine's precondition. Some authors object to that approach — their argument is that wait-conditions are part of synchronization specification and should be specified separately from preconditions that are part of functional specification [4][5]. Very often, wait-conditions are simply understood to be routine guards, as in other Eiffel-based concurrency models such as CEiffel [6] and CEE [4] where they are specified using a special syntax. We do not agree that wait-conditions should be treated as guards — they behave differently, in particular w.r.t. to inheritance and redefinition. Guards may be strengthened in a redefined feature; wait-conditions may only be weakened. That similarity of wait-

conditions and preconditions under inheritance was one of the arguments for using the wait-semantics as the generalized semantics for preconditions (see section 2.1).

The original SCOOP proposal [3] does not discuss the semantics of postconditions that involve separate calls. In the subsequent research on SCOOP [2][7] they are either assumed to have the same semantics as non-separate postconditions (i.e. they should hold after the execution of routine's body) or they are ignored. It is easy to demonstrate that both approaches are impractical — the former introduces potential for deadlocks and the latter simply excludes parts of postcondition from the reasoning rule (see section 2.2). We decided to apply the wait-semantics to postconditions in a way that unifies the treatment of separate and non-separate postcondition clauses and allows to rely on full postconditions when reasoning about the correctness of SCOOP programs.

In fact, we apply the wait-semantics to all assertions, including class invariants, checks, and loop assertions. This allows us to better understand the role of different assertions in a concurrent context and demonstrates that the traditional, sequential semantics is simply derived from the wait-semantics thanks to additional assumptions that can be made in a sequential context. Furthermore, it allows us to discover an interesting relation between correctness (safety) and liveness properties. We demonstrate that the notion of deadlock, traditionally related to liveness, can be formalized as assertion violation, traditionally viewed as a correctness issue.

The rest of this article is organized as follows. Section 2 describes the generalized semantics of contracts and refines the rule for reasoning about correctness of feature calls. Section 3 illustrates the use of new contract semantics with a producer-consumer example. Section 4 discusses the issues of deadlocks and run-time assertion checking. Section 5 discusses related work. Finally, section 6 concludes and describes future research directions.

2 Semantics of assertions

In this section, we propose a new semantics for contracts that is applicable in both concurrent and sequential contexts. The main motivation for this work is the observation that sequential computation (involving one processor) is a special case of concurrent computation (that may involve more than one processor); similarly, any synchronous call may be seen as a particular case of asynchronous call. Starting from that observation we make a similar claim concerning assertions. We say that every assertion has wait semantics; that semantics naturally reduces to the traditional (correctness) semantics if no concurrency is involved.

In the rest of this section, we proceed as follows. For each type of assertion, we first describe its traditional semantics, point out problems that arise in a concurrent context, and propose a generalized semantics. Secondly, we refine feature call rule 1.1 to take into account the new semantics. Finally, we demonstrate how the new semantics reduces to the traditional one thanks to additional assumptions that we can make in a sequential context. We discuss the semantics of preconditions and postconditions in

detail; other assertions are only described shortly since their new semantics is straightforward.

2.1 Preconditions

In SCOOP, preconditions may have two different meanings. Depending on whether they involve any separate calls, they are treated as correctness conditions or wait-conditions. Consider routine *store* in figure 1.

```
store (a_buffer: separate BOUNDED_QUEUE [INTEGER]; i: INTEGER)
            -- Store `i' in `a_buffer'.
    require
            not_full: not a_buffer.is_full
            i_positive: i > 0
    do
            . . .
    end
```

Figure 1. Feature with a separate precondition.

Clause *not_full* involves a call on separate target *a_buffer*, therefore it is a wait-condition. When a client calls feature *store*, the call will be blocked until that condition is satisfied. Clause *i_positive* does not involve any separate calls, therefore it is a correctness condition. When a client calls feature *store* and that clause is not satisfied, an exception is raised and the client is blamed for the contract violation. We can see three major problems with this solution.

First, the distinction between correctness and wait-conditions is based on the separateness of the involved calls. If the call target is declared as separate then the corresponding precondition clause has wait-semantics, even though, at run-time, the target might denote a non-separate object. Such a situation arises if feature *store* is called with a non-separate first actual argument. This is perfectly legal in SCOOP – the model disallows attachments from separate to non-separate entities but not the other way round [2]. We think that the correctness semantics should be applied in that case.

Second, it is sometimes necessary to transform a correctness condition into a wait-condition. Such need arises in the presence of inheritance. When redefining a feature, we are allowed to change the type of its formal arguments from non-separate to separate – such redefinitions are legal because clients of the ancestor class may still call the feature with non-separate actual arguments. It is alright to redefine the type of an argument to separate but what happens to precondition clauses that involve calls on that argument? They were correctness conditions in the original feature; they should become wait-conditions in the redefined feature. The necessity for wait-conditions to be considered as correctness conditions and vice-versa, as illustrated above, suggests that, in fact, both kinds of preconditions are equal and one semantics should be applied to them.

The third problem is that wait-conditions constitute no real contract between a client and a supplier. The supplier may assume that wait-conditions hold on entry but there is no obligation on the client to satisfy them. The client is only required to satisfy the non-separate part of the precondition. This is reflected in call rule 1.2 proposed in [2].

$$\frac{\{INV \wedge Pre_r\}\ \ body_r\ \ \{INV \wedge Post_r\}}{\{Pre_r^{nonsep}[\overline{a}/\overline{x}]\}\ \ r(\overline{a})\ \ \{Post_r^{nonsep}[\overline{a}/\overline{x}]\}} \qquad (1.2)$$

This tentative rule does not account for any potential interference of several processors. As a result, it cannot be used for proving program correctness – in particular, the client cannot be sure that the routine body will ever be executed.

We propose to adopt the following semantics. From the supplier's point of view, all preconditions preserve their correctness semantics, i.e. they are assumed to hold at the entry to the routine's body. For a client, all preconditions are wait-conditions, i.e. a non-satisfied precondition will force the client to wait until the precondition is satisfied. Conceptually, all precondition clauses, even non-separate ones, may cause waiting; in practice, the compiler and the run-time system may optimize the treatment of preconditions that do not involve separate calls – an exception will be raised if such assertions are violated.

According to the principles of Design by Contract, the obligation of satisfying the precondition is put on the client. Obviously, it is useless to require the client to ensure the precondition at the moment of the call since other clients may invalidate the precondition before the routine is executed (see section 1). Nevertheless, if we want to make sure that the precondition is satisfied when the routine starts executing, the client has to ensure that the precondition *eventually* holds. This is reflected in the refined call rule 1.3 (for the moment, ignore the part concerning the postcondition; we will discuss it and provide a full rule in section 2.2). *Acq* (*x*) stands for "x is acquired by current processor"; more precisely, it means that the processor which handles the object represented by *x* is locked for exclusive use by the current processor.

$$\frac{\{INV \wedge Pre_r\}\ \ body_r\ \ \{INV \wedge Post_r\}}{\{\Diamond(Acq(\overline{a}) \wedge Pre_r[\overline{a}/\overline{x}])\}\ \ r(\overline{a})\ \ \{Post_r[\overline{a}/\overline{x}]\}} \qquad (1.3)$$

In fact, the requirement put on the client is a bit stronger: *eventually, all actual arguments are acquired and the precondition holds.* In the subsequent discussion, we use temporal operators $\mathcal{U}$ ("until"), ◊ ("eventually"), and □ ("always"), as defined in [8]. The use of the "eventually" operator in rule 1.3 is essential to capture the intended semantics of a feature call – the client may wait but not infinitely. Let us see how this semantics can be applied to our example routine *store* from figure 1. A client executing *store* (*buffer*, 10) has to ensure that

$\Diamond$ (*Acq* (*buffer*) $\wedge$ $\neg$*buffer.is_full* $\wedge$ 10 > 0)

holds before the call. The call will be postponed until *Acq* (*buffer*) and both precondition clauses are true.

Note that non-satisfiability of a precondition clause results in the client waiting forever. For example, a client calling *store* (*buffer*, -5) cannot ensure the required property because $\neg$(*-5 > 0*). This means that the client will be stuck forever. But it is obvious that, in that particular case, waiting for the precondition does not make any sense because -5 will never become greater than 0. We can decide immediately that the precondition will never be satisfied; the run-time system may react appropriately by raising an exception. In fact, in a situation when the client waits, all properties of non-separate objects are invariant. That is, for a non-separate x, if property *P* (*x*) is true, then it will *always* remain true; conversely, if *P* (*x*) is false, then it will *never* become true. Thanks to that invariance, we can conclude that

$$P(x) \Longleftrightarrow \Box P(x) \Longleftrightarrow \Diamond P(x) \qquad (1.4)$$

Applying rule 1.4 to property $\neg$(*-5 > 0*) we can prove

$\Box\neg$(*-5 > 0*), hence
$\neg \Diamond$(*-5 > 0*), hence
$\neg \Diamond$ (*Acq*(*buffer*) $\wedge$ $\neg$*buffer.is_full* $\wedge$ -5 > 0)

So, a call to *store* (*buffer*, -5) that conceptually should be blocked forever, will result in an exception rather than an infinite wait. Let us consider a situation where all actual arguments are non-separate:

non_separate_buffer: *BOUNDED_QUEUE* [*INTEGER*]
. . .
store (*non_separate_buffer*, 10)

Acq (*non_separate_buffer*), **not** *non_separate_buffer.is_full*, and 10 > 0 are all properties of non-separate objects. By applying rule 1.4 and taking into account the fact that *Acq* (*x*) holds trivially for any non-separate *x* (because *x* is handled by the same processor as **Current**) we can simplify the client's obligation to

$\neg$*buffer.is_full* $\wedge$ *10 > 0*

which is precisely the traditional (sequential) precondition. As you can see, thanks to additional assumptions that can be made about the properties of non-separate objects, wait-condition semantics reduces nicely to the usual correctness semantics when no concurrency is involved. Indeed, rule 1.3 applied to sequential code reduces to rule 1.1 (in section 2.2 we will show that the postcondition part can be reduced following the same approach).

2.2 Postconditions

The treatment of postconditions in SCOOP is unsatisfactory. The initial design of SCOOP assumed that postconditions involving separate calls could be treated as correctness conditions and it did not develop the topic any further. Obviously, the evaluation of a separate postcondition may introduce delays due to the asynchronous nature of separate calls; such postconditions certainly cannot be treated in the same way as non-separate ones. We considered three ways of dealing with the problem:

- prohibit separate postconditions,
- allow separate postconditions but ignore them in the proof rule (see rule 1.2) and do not evaluate them at run-time,
- require that routine blocks until separate postconditions hold.

The first two proposals are not real solutions because they restrict the practical use of postconditions to non-separate ones only. The third proposal is interesting because it allows reasoning about concurrent code using rule 1.3. The client gets the guarantee that the call will terminate in a state that satisfies the postcondition. Unfortunately, blocking until all postconditions are satisfied is very inefficient and may lead to deadlocks, in particular in the presence of callbacks. Consider feature *spawn_two_activities* in figure 2.

```
spawn_two_activities (location_1, location_2: separate LOCATION)
        -- Launch jobs at `location_1' and `location_2'.
    do
        location_1.do_job
        location_2.do_job
    ensure
        location_1.is_ready
        location_2.is_ready
    end
```

Figure 2. Separate postconditions.

A client executing a call to *spawn_two_activities* (*york*, *tokyo*) does not want to wait until the job is done at both locations – in particular if one of these locations terminates much later than the other. In fact, the client does not want to wait at all. Still, it wants to have some guarantee about the job being done. Such guarantees are naturally expressible as postconditions but, as we can see here, waiting for all postconditions does not solve the problem. Additionally, a situation where one of the locations tries to call back the client (or call the other location) results in a deadlock. The client cannot release the locks before the postconditions are evaluated; the supplier needs to acquire one of the locks held by the client in order to establish the postcondition. So, the client waits for the supplier while the supplier waits for the client — they end up in a deadlock.

When does a client really need the postcondition to hold? In figure 3, the client spawns two activities in York and Tokyo, does some local work, and asks for results of remote activities.

york, tokyo: **separate** *LOCATION*
. . .
spawn_two_activities (*york, tokyo*)
do_local_stuff
get_result (york)
do_local_stuff
get_result (tokyo)
. . .

Figure 3. Concurrent activities.

The client should not wait after the execution of *spawn_two_activities* (*york, tokyo*) but continue with the execution of its local activity (*do_local_stuff*). Only at the moment when it executes *get_result (york)* should the postcondition clause *york.is_ready* matter – we may expect that the precondition of *get_result* depends on the postcondition of *spawn_two_activities*. In other words, the call to *get_result* should not proceed unless the postcondition *york.is_ready* holds. Note that, at that moment, it does not matter whether the other activity (in Tokyo) has terminated successfully. The client is not (yet) interested in it. Assume that *york.is_ready* holds and the client can execute *get_result (york)* followed by some local activity (*do_local_stuff*). The execution of *get_result (tokyo)* depends on the postcondition clause *tokyo.is_ready* – it is only now that the client becomes interested in that postcondition. We can observe that the postcondition clause concerning *york* does not matter anymore. In fact, the state of *york* might have changed as a result of call to *get_result (york)*. This simple example suggests that:

- it is not necessary for a separate postcondition to hold immediately after the execution of the routine's body,
- wait-semantics applies to postconditions but waiting happens on the supplier side – the separate target is not released until the postcondition clause is satisfied,
- individual postcondition clauses should be considered independently.

Let us try to formalize this way of reasoning and refine the call rule by introducing temporal operators that capture the intended semantics.

$$\frac{\{INV \wedge Pre_r\}\ body_r\ \{INV \wedge \forall_i \Diamond Post_r^i\}}{\{\Diamond(Acq(\overline{a}) \wedge Pre_r[\overline{a}/\overline{x}])\}\ r(\overline{a})\ \{\forall_i(\Diamond Rel(a^i) \wedge \neg Rel(a^i)\ \mathcal{U}\ Post_r^i[\overline{a}/\overline{x}])\}} \qquad (1.5)$$

Rule 1.5 weakens the obligation on the routine's implementor so that the body only has to ensure that *the invariant holds immediately and each postcondition clause holds eventually*. *Rel* (*x*) stands for "x is released"; more precisely, it means that the processor which handles the object represented by *x* is unlocked, provided that it is

not the current processor (if it is, then *Rel* (*x*) holds vacuously). We can express it as: *Rel* (*x*) = ¬*Acq* (*x*) for all *x* denoting separate objects; *Rel* (*x*) = true for all *x* denoting non-separate objects. $Post^i_r$ denotes i-th postcondition clause of *r*. For example, $Post^1_{spawn\_two\_activities}$ corresponds to *location_1.is_ready*. Similarly, $Post^2_{spawn\_two\_activities}$ corresponds to *location_2.is_ready*. The guarantees for the client should be read as follows: *for all postcondition clauses, arguments involved in the given postcondition clause are eventually released but not until that postcondition clause holds*. a^i denotes the set of arguments that are involved (serve as call target) in postcondition clause $Post^i_r$. The weakening of obligations put on the routine's body is reflected in the redefined notion of local correctness.

Definition 2. Local correctness. *Routine r of class C is locally correct iff after the execution of r's body, class invariant* Inv_C *holds and each postcondition clause will hold eventually, provided that both the invariant and the precondition* Pre_r *were fulfilled before the body started executing.*

If we apply rule 1.5 to the call *spawn_two_activities* (*york*, *tokyo*) we obtain the following obligation on the routine body:

$$\Diamond\, location\_1.is\_ready \;\wedge\; \Diamond\, location\_2.is\_ready$$

which, supposedly, can be simply proved using the postcondition of feature *do_job* used in the body of *spawn_two_activities*. The guarantee given to the client is:

$$\Diamond\, Rel\,(york) \wedge (\neg\, Rel\,(york)\ \mathcal{U}\ york.is\_ready)$$

$$\wedge\ \Diamond\, Rel\,(tokyo) \wedge (\neg\, Rel\,(tokyo)\ \mathcal{U}\ tokyo.is\_ready)$$

We can use that guarantee to satisfy the requirement of the subsequent calls to *get_result (york)* and *get_result (tokyo)*. In some sense, that new semantics of postconditions offers the same guarantees but "projected" into the future. The client is interested in establishing each postcondition clause at the moment when the involved objects are released. We think that such semantics captures the intended meaning of postconditions in the presence of asynchrony. It gives more flexibility in programming by removing the unnecessary waiting; at the same time, it makes sure that all postconditions constitute a contract between clients and suppliers, so that it is possible to reason about feature calls using a simple rule.

According to our semantics, the non-satisfiability of a postcondition clause results in the involved objects being held forever. These objects will never be released so they can never be acquired again by any client. Therefore, a violated postcondition may result in a deadlock. In practice, if the involved objects are non-separate, we can use additional assumptions (rule 1.4) to solve the problem and react to such a situation by raising an exception rather that waiting forever. Recall that all properties of non-separate objects are preserved while the client is waiting. Similarly to *Acq* (*x*), also *Rel* (*x*) is trivially true for all non-separate *x*. Therefore, if postcondition clause $Post^k_r$

that does not involve any separate calls does not hold when routine *r* terminates, an exception is raised and the supplier is blamed for the contract violation. Conceptually, though, a violated postcondition clause results in infinite waiting.

Finally, as a "sanity check", let us demonstrate that in a sequential context rule 1.5 reduces to the standard rule for sequential programs (1.1). We already demonstrated in section 2.1 that the precondition part of rule 1.5 reduces to the corresponding part of 1.1. Here, we focus on postconditions. From rule 1.4 and $INV \wedge \forall_i \Diamond Post_r^i$ we obtain $INV \wedge \forall_i Post_r^i$ that can be further simplified to $INV \wedge Post_r$ which is precisely the obligation on the routine's body in rule 1.1. On the client's side, since $Rel(a^i)$ is true for all *i*, the guarantee may be simplified to

$$\forall_i (\textbf{true} \wedge (\textbf{false } \mathcal{U} \ Post_r^i [a/x]))$$

and, using the property of the temporal operator *until*, to $\forall_i Post_r^i [a/x]$ and finally to $Post_r [a/x]$ which is precisely the guarantee given to the client by rule 1.1.

We mentioned earlier that the previous proposal – blocking until all postconditions are satisfied (see rule 1.3) – may lead to deadlocks if separate calls in a routine body involve cross-calls, i.e. when one separate supplier needs to access another one. How does our approach deal with such situations? Consider again the situation depicted in figure 2. Assume that the activity spawned at location *york* (routine *do_job*) needs to access location *tokyo* and perform some operations on it. Certainly, *tokyo* will not be released by the client (and thus become available to other clients) until the postcondition clause *tokyo.is_ready* is satisfied. So, *york*'s call will be blocked until then. On the other hand, the client will not be blocked because it does not need the access to *tokyo* or *york* to continue its local activity (*do_local_stuff*). When *tokyo* is released, *york*'s call will lock it, perform the necessary calls, and release it again. Now, postcondition clause *york.is_ready* is satisfied and *york* is released. Our client, which by that time has probably finished its local activity and is waiting for *york* to become available, can now execute *get_result (york)*. As you can see, thanks to the new semantics of postconditions, it is possible to use postconditions even in the presence of cross-calls. Note that, in our example, a callback to the client would still result in a deadlock (in [9] we propose a lock passing mechanism that allows to avoid such deadlocks). No problem would arise if the client did not try to perform any calls to *york* after the first call to *spawn_two_activities*. In such a situation, *york*'s callback would simply block until the client becomes idle (i.e. it is released by its own client), and then proceed.

Discussion

Rule 1.5 is not strong enough to allow for fully compositional proofs of correctness and liveness. The following example, due to Jonathan Ostroff, illustrates the problem. Let us reconsider the York–Tokyo scenario in figure 3. Suppose that, when calling *spawn_two_activities*, we can show that we eventually acquire *york* and *tokyo* resources as required by (1.5). The rule then informs us that eventually the postcondi-

tions of *spawn_two_activities* will be satisfied; only after that will the resources be ceded to other putative processors. However, in our example, all this might happen before the call to *get_result* (*york*) as *do_local_stuff* may take a long time. In the meantime other clients (handled by a different processor) may invoke routines that could change the state of *york*. Thus, by the time we get to *get_result* (*york*), the postcondition of *spawn_two_activities* may no longer hold; as a result, our call to *get_result* (*york*) may not proceed.

The problem appears to be that the postcondition of *spawn_two_activities* is not projected sufficiently far into the future (which would be required to get rid of the need for global reasoning). In this case, we need to apply global reasoning (as illustrated in the producer-consumer example in section 3) to show that there are no other clients that could change the postconditions. Hence, we would need a combination of local and global reasoning to use the postcondition of *spawn_two_activities* for *get_result* (*york*). Nevertheless, if the concerned resource (here *york*) is guaranteed to be exclusively used by our client (i.e. it is locked on behalf on our client in the context of the routine where both calls are executed), local reasoning is sufficient. The fact that the postcondition of *spawn_two_activities* does not hold immediately is irrelevant here — we may still use it to show that the precondition of *get_result* (*york*) will hold when its body is executed (because no call on *york* present in the body of *get_result* may start executing before all previous calls on *york* have terminated). The results of our recent work [10] show that, in such cases, we can even get rid of temporal operators and use a simpler rule than (1.5) for reasoning about asynchronous feature calls. Global reasoning (using Ostroff et al.'s method [11]) is only necessary for calls that acquire additional (fresh) resources. We hope that such a combination of local and global reasoning will allow for local proofs of partial correctness and it may be used to prove library classes without the need to know the context in which they are utilised; on the other hand, proofs of total correctness (that is partial correctness + termination + absence of deadlocks) will require global reasoning.

2.3 Invariants

Invariants play a very important role in the Design by Contract methodology. They are the primary tool for ensuring the consistence of objects. To prove local correctness of a routine, we may assume the invariant before the execution of the body and we have to guarantee that it holds again when the body terminates. Note that our refined rule for feature calls (1.5) follows that pattern; it does not introduce any temporal operators that would suggest a different semantics of invariants. This might be a bit surprising since we started this paper with the claim that wait-semantics is the natural semantics for all assertions.

A closer look at SCOOP rules explains why we apply the traditional (correctness) semantics to invariants. SCOOP's *separate call rule* requires that the target of a separate call must appear as formal argument of the enclosing routine. But calls appearing in invariants have no enclosing routines! Therefore, it is prohibited to use separate calls in invariants. Conceptually, we still consider that a violated invariant causes

waiting but in practice, since all its clauses only contain non-separate calls, we may use rule 1.4 to reduce the wait-semantics to the correctness semantics. As in the case of preconditions and postconditions, the run-time system is able to react to a violated invariant by raising an exception.

2.4 Other assertions

We apply the wait-semantics to other assertions: checks, loop variants, and loop invariants. Conceptually, a violated assertion causes infinite waiting but in practice waiting only happens if the assertion involves a separate call — in that case the client needs to wait for the result. When an assertion has been evaluated and it does not hold, an exception is raised. We can consider that wait-semantics of such assertions always reduces to correctness semantics because rule 1.4 also applies to separate objects locked in the current context.

Consider the loop in feature *remove_one_by_one* in figure 4. The assertions capture the essence of that loop – at every step, the number of elements in *a_list* is reduced, and the number of elements that remain plus the number of elements already removed correspond to the initial number of element. Because *a_list* may denote a separate object, the evaluation of *a_list.count* may cause waiting. So, both the loop invariant and the loop variant may cause waiting. On the other hand, as soon as an assertion has been evaluated and it does not hold, its violation results in an exception.

```
remove_one_by_one (a_list: separate LIST [G])
        -- Remove all elements of `a_list' one-by-one.
    local
        initial, removed: INTEGER
    do
        from
            initial := a_list.count
            a_list.start
        until
            a_list.is_empty
        invariant
            a_list.count + removed = initial
        variant
            a_list.count
        loop
            a_list.delete
            removed := removed + 1
        end
    ensure
        a_list.is_empty
    end
```

Figure 4. Separate loop assertions.

3 Producer – consumer example

In this section, we show how the new feature call rule 1.5, based on the proposed wait-semantics of preconditions and postconditions, can be used for reasoning about the correctness of SCOOP programs. We use a simple producer-consumer scenario depicted in figure 5. Implementation of producers and consumers is given in the Appendix (figures 6 and 7, respectively). We assume that the size of the buffer is greater than 0 and that the buffer is bounded. We chose to consider just one producer and one consumer because this allows us to ignore assumptions about the scheduling policy of SCOOP — we are able to prove the correctness of our example, including the absence of deadlock and starvation, even without relying on the fairness guarantees of SCOOP's scheduler. To prove absence of starvation in a scenario with *n* producers and *m* consumers we would need to assume the FIFO scheduling policy of SCOOP.

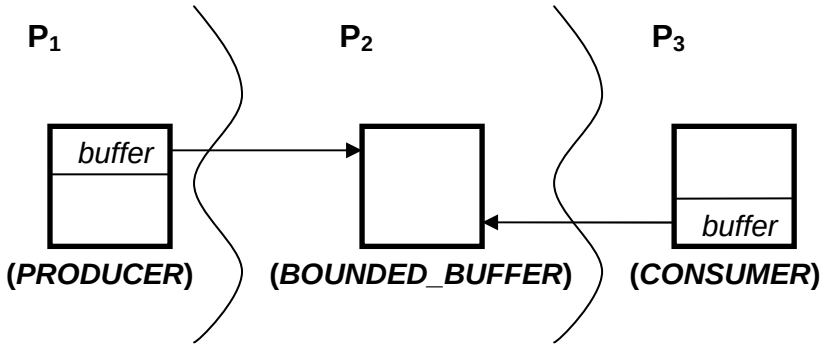

Figure 5. Producer-consumer scenario.

Producer and consumer objects exhibit very similar activity. Essentially, they execute an infinite loop, accessing the shared *buffer* at each loop step. Producer accesses *buffer* via a call to *store* (*buffer*, 10); consumer uses a call to *retrieved* (*buffer*) for that purpose. Both features are equipped with precise (although not exhaustive) contracts. *store* requires that *buffer* be not full and ensures that the number of elements in *buffer* increase by 1. *retrieved* requires that *buffer* be not empty and ensures that the number of elements in *buffer* decrease by 1. We assume that contracts of features *put* and *remove* in class *BOUNDED_QUEUE* [*G*] correspond to the contracts of *store* and *retrieved*, respectively. We want to use rule 1.5 for proving the correctness of calls to *store* and *retrieved*. The first step is to show that these features are locally correct according to Definition 2. For *store*, we need to prove

{¬*a_buffer.is_full*}
    a_buffer.put (*i*)
{◊ *a_buffer.count* = **old** *a_buffer.count* + 1}

This is straightforward, given the precondition and the postcondition of *put*.

Similarly, for *retrieved*, we need to prove

{¬*a_buffer.is_empty*}
Result := *a_buffer.item*
a_buffer.remove
{◊ *a_buffer.count* = **old** *a_buffer.count* - 1}

Once again, the proof is straightforward because we can rely on the contracts of *item* and *remove*. Note that the *eventually* operator (◊) is essential here – it would be impossible to prove that the postcondition holds immediately after the execution of the body of *retrieved*.

We established local correctness of *store* and *retrieved*. Let us now apply rule 1.5 to prove correctness of calls to these routines. For the producers's call *store* (*buffer*, 10) we need to show that

◊ (*Acq* (*buffer*) ∧ ¬*buffer.is_full*)

holds before the call. We prove it by case analysis on the state of *buffer*.

Case 1. *buffer* is idle and *buffer.is_empty* holds. Therefore, the consumer cannot get hold of *buffer*; the producer can immediately establish

Acq (*buffer*) ∧ ¬*buffer.is_full*

hence

◊ (*Acq* (*buffer*) ∧ ¬*buffer.is_full*)

and we are done.

Case 2. *buffer* is idle and *buffer.is_full* holds. The producer cannot get hold of buffer but

buffer.is_full ∧ *buffer.size* > 0 ⇒ ¬*buffer.is_empty*

Therefore, the consumer will eventually execute a call to *retrieved*, and thus establish ¬*buffer.is_full*. It results in **case 1** if the size of *buffer* is 1; otherwise, in **case 3**.

Case 3. *buffer* is idle and ¬*buffer.is_full* and ¬*buffer.is_empty* hold.
Either (a) the producer acquires buffer, in which case

Acq (*buffer*) ∧ ¬*buffer.is_full*

holds, and so does

◊ (*Acq* (*buffer*) ∧ ¬*buffer.is_full*)

and we are done, or (b) the consumer acquires *buffer*, in which case the consumer executes a call to *retrieved*. As a result, we are back to **case 3** or **case 1**.

Case 4. *buffer* is not idle.
There may be only two reasons for that: either (a) *buffer* has not been released yet after the previous call to *store*, or (b) *buffer* has not been released yet after the call to *retrieved*. In both cases, *buffer* will be eventually released: we can assume that from

rule 1.5, given that both features are locally correct as demonstrated above. As a result, we will eventually be back to **case 1**, **case 2**, or **case 3**. □

Note that we do not rely on any particular scheduling policy here. In **case 3**, we do not know who will proceed first. Nevertheless, even if we assume a very unfair policy, e.g. the consumer overtakes the producer, we eventually hit **case 1** where only the producer is allowed to proceed. We use similar analysis for the consumer's call *retrieved* (*buffer*). We need to show that

$$\Diamond\,(\,Acq\,(buffer)\ \wedge\ \neg buffer.is\_empty\,)$$

holds before the call.

Case 1. *buffer* is idle and *buffer.is_empty* holds. Since

$$buffer.is\_empty \wedge buffer.size > 0\ \Rightarrow\ \neg buffer.is\_full$$

the producer will execute a call to *store*, and thus establish $\neg buffer.is\_empty$. It results in **case 2** if the size of *buffer* is 1; otherwise, in **case 3**.

Case 2. *buffer* is idle and *buffer.is_full* holds.
We can immediately establish

$$Acq\,(buffer) \wedge \neg buffer.is\_empty$$

and so

$$\Diamond\,(\,Acq\,(buffer)\ \wedge\ \neg buffer.is\_empty\,)$$

and we are done.

Case 3. *buffer* is idle and $\neg buffer.is\_full$ and $\neg buffer.is\_empty$ hold.
Either (a) the consumer acquires *buffer*, in which case

$$Acq\,(buffer) \wedge \neg buffer.is\_empty$$

holds, and so does

$$\Diamond\,(\,Acq\,(buffer)\ \wedge\ \neg buffer.is\_empty\,)$$

and we are done, or (b) the producer acquires *buffer*, in which case the producer executes a call to *store*. As a result, we are back to **case 3** or **case 2**.

Case 4. *buffer* is not idle.
Idem as for *store*. □

From rule 1.5 we can now conclude that after a call to *store* (respectively *retrieved*) *buffer* is eventually released in a state that satisfies the postcondition. The use of rule 1.5 is certainly much more complex than reasoning about sequential programs – the latter is based on a simpler rule 1.1 that does not involve any temporal operators. Obviously, the rule for concurrent programs must take into account the potential interference of several processors, hence the complexity of reasoning. On the other hand, we are able to prove the absence of deadlock using the same rule. So, the increased complexity pays off – concurrent code that is proved correct is also deadlock-free.

4 Discussion

Deadlocks

Absence of deadlocks is one of the most interesting properties of concurrent programs. In fact, the problem of deadlocks was one of the initial motivations of our work. We set off to devise a methodology for deadlock prevention, detection, and resolution in SCOOP programs. The first step towards developing such a methodology is to understand the relation between deadlocks and contracts. Traditionally, contracts are used for enforcing correctness (safety) properties; a separate proof is needed for liveness properties such as absence of deadlock or starvation. When discussing the new semantics of assertions, we mentioned that an *assertion violation results in a deadlock*, at least conceptually. Let us push this argument a bit further and claim the opposite relation, i.e. *every deadlock corresponds to a violated assertion.*

In which situation can a deadlock happen? Consider again the sequence of calls in figure 2. The first possibility is that either of separate objects *york* or *tokyo* can never be acquired by the client. But this means that either ◊*Acq* (*york*) or ◊*Acq* (*tokyo*) does not hold, so the client's obligation, as expressed in rule 1.5, is violated. One may claim that *Acq* (*york*) is not really part of an assertion (in this case a precondition) because locking of arguments in SCOOP is based on argument passing. Nevertheless, we may assume that, for every separate formal argument *x* that appears in the signature of a feature there is an implicit precondition *Acq* (*x*) and that locking is based on preconditions only. Note that the most common deadlock situation, i.e. *a* tries to lock *b*, *b* tries to lock *c*, *c* tries to lock *a*, corresponds to that first possibility. The second possibility is that both *york* or *tokyo* can be eventually acquired by the client but, whenever they can be acquired, their state does not satisfy the precondition. This kind of deadlock is also caused by the client's inability to satisfy the precondition. The third possibility is a postcondition violation. Assume that postcondition *york.is_ready* of *spawn_two_activities* cannot be satisfied. According to our semantics, *york* is not released until all postcondition clauses that involve it are satisfied. Therefore, *york* is never released. This does not cause a deadlock by itself but a deadlock will happen as soon as some client tries to acquire *york* (as our client does using a call to *get_result*).

As demonstrated in section 2, the violation of an assertion that does not involve any separate calls also results in a deadlock albeit only conceptually – in practice, there is no need to wait forever because the violation can be detected immediately and an exception can be raised. This also applies to invariants, checks, and loop assertions.

Exception handling

In the previous sections we often mentioned run-time exceptions. The asynchronous nature of some feature calls makes it impossible to rely on standard exception handling. For example, it is not always possible to propagate an exception to a client because the client might have already left the context of the enclosing routine. Therefore, we need some support for asynchronous exceptions. Exception handling is beyond the scope of this paper; we assume that an appropriate mechanism for handling

asynchronous exceptions is available. Such a mechanism has been recently proposed by Arslan et al. [12].

Assertion checking at run-time

Meyer [2] mentioned the problem of run-time assertion checking in a concurrent context. He concluded that "The assertions are an integral part of the software, whether or not they are enabled at run time. Because in a correct sequential system the assertions will always hold, we may turn off assertion checking for efficiency if we think we have removed all the bugs; but conceptually the assertions are still there. With concurrency the only difference is that certain assertions – the separate precondition clauses – may be violated at run time even for a correct system, and serve as wait conditions. So the assertion monitoring options must not apply to these clauses."

We claim that assertion checking may be turned off even for wait-conditions. To demonstrate it, let us first see under what circumstances run-time checking of non-separate assertions may be turned off. Following rule 1.1, if for all calls the client can ensure that the precondition of the called routine is satisfied ($Pre_r[a/x]$ holds immediately), then precondition checking may be turned off. If we can demonstrate that the corresponding assertion in rule 1.5, i.e. $Acq\ (a) \wedge Pre_r[a/x]$ holds immediately (note the absence of temporal operator $\Diamond$) then we may also turn off precondition checking in a concurrent context. But this means that we can only do it if separate objects are immediately available. We can actually weaken that assumption a bit and only require that, at the moment of the call, they become eventually available and, whenever they are available, the precondition holds:

$$\Diamond Acq\ (a) \wedge\ (\ Acq\ (a) \Rightarrow Pre_r[a/x]\ )$$

If we can demonstrate that this is satisfied for all calls to a particular feature then precondition checking for that feature may be turned off but clients may still wait for objects that they try to acquire.

Postcondition checking may be turned off in a sequential context if every routine is locally correct. The same applies in a concurrent context, although we use a weaker notion of local correctness (Definition 2). Invariant checking follows the same rules as in a sequential context; so does checking of other assertions.

5 Related work

Bailly [13] proposes an operational semantics for a subset of SCOOP and a gives a set of rules for the inference of safety properties of concurrent programs. The author assumes a different semantics of separate preconditions – they are merely guards of conditional critical regions represented by routine bodies. Guards are excluded from contracts and treated separately from traditional (correctness) preconditions. The approach does not support inheritance, therefore problems caused by guard strengthening vs. precondition weakening are not discussed. The treatment of postconditions

is identical in the concurrent context, although the author comments on the infeasibility of formal reasoning with rule 1.2. Following the CCR semantics, unlocking of separate objects locked by a given routine is performed atomically. As a result, it is impossible to reason about features that involve separate callbacks; Additionally, query calls may only appear at the end of a routine's body. This is in stark contrast to our approach of individual unlocking that does not impose any restrictions on routine bodies. Bailly also proposes a non-compositional proof system along the lines of the proof system for concurrent Java programs [14]. Unlike in Java, no interference-freedom test is required in SCOOP because intra-object concurrency is prohibited; on the other hand, the presence of asynchronous calls increases the complexity of the proof system.

Sutton [15] describes a new strategy for condition-based process execution, based on a delayed evaluation of preconditions and postconditions. Although preconditions have guard semantics, they are evaluated in parallel with tasks; a task might be allowed to execute even though some of its preconditions have not been evaluated yet. They only have to hold at a particular point of the task's execution; otherwise, the task is put on hold or cancelled. Similarly, a task may terminate even though some of its postconditions have not been established yet. Nevertheless, they have to be established eventually; otherwise, the task must be cancelled (rolled back or compensated, since tasks are transaction-like in that framework). The postcondition semantics is very similar to ours, except that in our approach a violated postcondition results (at least conceptually) in a deadlock. The precondition semantics proposed by Sutton is different but it may be simulated in our model simply by splitting up a task (enclosing routine) into smaller sub-tasks where all subtasks require their preconditions to hold right on entry to their bodies.

Rodriguez et al. [16] propose a concurrent extension to JML where method guards are treated in a similar way as Sutton's preconditions. Guards are specified in feature headers, after preconditions. If a feature is called in a state where the guard does not hold, the feature does not always block – the guard does not need to hold at the beginning of the body but only at a point marked with a special statement label *commit*. If no commit point is specified, it is implicitly assumed at the end of the body. Guards are simply predicates that have to be satisfied at the commit point but no implicit waiting is involved – it is up to the programmer to implement it, e.g. in the form of a busy-waiting loop. Therefore, we can view guards as a help in the static verification of atomicity properties but they certainly do not facilitate the construction of concurrent programs – programmers are forced to write explicit synchronization code. This inevitably leads to inheritance anomalies. From that point of view, the generalized semantics of pre-conditions that we propose provides a safer and more convenient support for synchronization.

Several concurrent extensions of Eiffel, such as CEiffel [6], CEE [4], Distributed Eiffel [17] use guard-based synchronization. Unlike in SCOOP, guards are specified using a different syntax than preconditions. Syntactic separation of preconditions and guards facilitates programming; unfortunately, in all three approaches, guards are not part of routine contracts and they are not used for formal reasoning.

The SCOOP-to-Eiffel-Generator (SECG) [18] relies on wait semantics for separate preconditions. SECG translates SCOOP code into pure Eiffel code with embedded calls to threading library EiffelThread. Separate preconditions are always treated as guards because objects represented by separate entities are assumed to be indeed separate w.r.t. the client. If attachments from non-separate to separate entities were allowed, a precondition violation might lead to a deadlock. In our approach, such deadlocks are only conceptual; in practice, the run-time is able to detect the assertion violation and react by raising an exception. SECG implements atomic lock release and treats separate postconditions just like non-separate ones; again, this may lead to problems discussed in section 2.2.

Traditionally, proof methods for concurrent programs are non-compositional, i.e. it is necessary to consider the whole program in order to prove correctness of its parts [19]. This also applies to our approach: in general, we cannot prove a single class without knowing the code of all its clients and suppliers. It would be interesting to look for a compositional method for reasoning about SCOOP programs. In his PhD dissertation [20], Jones describes a compositional approach to proving correctness properties of concurrent shared-memory programs. He enriches contracts with two additional assertions – *rely* and *guarantee* – that represent assumptions on (respectively commitment to) the environment of a process. Compositional reasoning is made possible through the use of these assertions together with standard preconditions and postconditions. Unfortunately, rely-guarantee specifications may only be applied to shared-memory models with no aliasing. Nevertheless, similar approaches (assumption-commitment) for message-passing systems have also been proposed [21]. These are more appropriate for SCOOP-like models that are based on asynchronous feature calls. An interesting survey of research efforts related to compositional approaches for concurrency is [22]. The results of our recent work on proofs for concurrent programs [10] suggest that it is possible to achieve a high degree of modularity in proofs of concurrent object-oriented programs; nevertheless, some proofs of still require global reasoning. A fully modular proof system for SCOOP would require much more expressive contracts: new types of assertions would be necessary to capture the locking behavior of routines (i.e. what additional resources a routine may request during the execution of its body) and their frame properties. These might be viewed as a particular case of assumption-commitment specifications.

6 Conclusions and future work

We proposed a generalized semantics for contracts that is applicable in concurrent and sequential contexts. Our methodology does not discriminate between (sequential) preconditions and (separate) wait-conditions; we give a simple semantics to preconditions that caters for the needs of concurrency and nicely reduces to the sequential semantics when no concurrency is involved. This is an important improvement w.r.t. the original SCOOP model where wait-conditions were essentially “hijacked” preconditions, much closer to the concept of *guards* (this also raised the problem of wait-condition weakening vs. guard strengthening). We have also defined a new semantics

for postconditions that relies on independent evaluation of individual postcondition clauses. Compared with the original SCOOP, our semantics allows for the use of separate calls in postconditions without the danger of deadlocking. Also, it makes it possible to reason about features that involve separate callbacks.

We used the new semantics to define a rule for reasoning about the correctness of feature calls. The rule (1.5) is a generalization of the sequential call rule (1.1). It relies on routine contracts but also reflects the interference of several parallel activities inherent in every concurrent system. We think that this rule captures the intended semantics of SCOOP and it lays a solid basis for a future development of a full-fledged proof system for concurrent object-oriented programs.

A (surprising at first) by-product of this research was the formalization of deadlocks as assertion violations. We demonstrated that deadlocks result from non-satisfiable contracts. This conclusion also led to a deeper understanding of the rôle of assertions in a program: we showed that they are an integral part of software and they cannot be simply ignored at execution. We defined precise conditions under which assertion checking may be turned off.

We are currently working on a full formalization of SCOOP, including an operational semantics and proofs of type safety. This formal model is based on the new semantics of contracts and takes into account further extensions of the model, such as an ownership-like type system for reasoning about object locality [23][24] and a refined locking policy that allows for precise specification of locking requirements and introduces a lock-passing scheme [9]. We are planning to implement a support for generalized contracts in the next release of our *SCOOPLI* library and *scoop2scoopli* tool. So far, the new semantics of preconditions, invariants, checks, and loop assertions has been implemented; our next step will be the implementation of postconditions.

We are interested in devising a modular proof system for SCOOP programs. The results of our recent work [10] show that, in many cases, we can get rid of temporal operators and use standard Hoare rules for reasoning about asynchronous feature calls; as a result, we can achieve a higher degree of modularity.

7 References

1. Meyer, B.: *Applying "Design by Contract"*, in IEEE Computer Volume 25, 1992, pp. 40–51.
2. Meyer, B.: *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, 1997.
3. Meyer, B.: *Systematic Concurrent Object-Oriented Programming*, in Communications of the ACM, Volume 36, Number 9, September 1993, pp. 56-80.
4. Jalloul, G.: *Concurrent object-oriented systems: a disciplined approach*, PhD thesis, University of Technology, Sydney, Australia, June 1994.
5. Caromel, D.: *Towards a Method of Object-Oriented Concurrent Programming*, in Communications of the ACM, Volume 36, Number 9, September 1993, pp. 90-102.
6. Löhr, K.-P.: *Concurrency annotations for reusable software*, Communications of the ACM, Volume 36. Number 9, 1993, pp. 81–89.

7. Nienaltowski P., Arslan V., Meyer B.: *Concurrent object-oriented programming on .NET*, IEE Proceedings Software, Special Issue on ROTOR, October 2003.
8. Manna, Z, Pnueli, A.: *The temporal logic of reactive and concurrent systems*, Springer-Verlag, New York, 1992.
9. Nienaltowski, P.: *Flexible locking in SCOOP*, First International Symposium on Concurrency, Real-Time and Distribution in Eiffel-like Languages (CORDIE), 4-5 July 2006, York, UK.
10. Nienaltowski, P., Meyer, B., Ostroff, J.S.: *Reasoning about concurrent object-oriented programs*, (to be submitted).
11. Ostroff, J., Torshizi, F.A., Feng Huang, H.: *Verifying Properties beyond Contracts of SCOOP Programs*, First International Symposium on Concurrency, Real-Time and Distribution in Eiffel-like Languages (CORDIE), 4-5 July 2006, York, UK.
12. Arslan, V.: *Asynchronous exceptions in concurrent object-oriented programming*, First International Symposium on Concurrency, Real-Time and Distribution in Eiffel-like Languages (CORDIE), 4-5 July 2006, York, UK.
13. Bailly, A.: *Formal semantics and proof system for SCOOP*, technical report, available at http://se.ethz.ch/research/scoop.html.
14. Abraham, E., de Boer, F.S., de Roever, W.P., Steffen, M.: *An assertional proof system for multithreaded Java*, in special issue of TCS, Volume 331, 2004, pp. 251-290.
15. Sutton, S. M.: *Preconditions, postconditions, and provisional execution in software processes*, Technical report 95-77, Computer Science Department, University of Massachusetts, July 1995.
16. Rodriguez, E., Dwyer, M., Flanagan, C., Hatcliff, J., Leavens, G. T., Robby: *Extending JML for modular specification and verification of multi-threaded programs*, in European Conference on Object-Oriented Programming (ECOOP), July 2005, pp. 551–576.
17. Gunaseelan, L., LeBlanc, R.J.: *Distributed Eiffel: A language for programming multi-granular objects*, in Proceedings of the 4th International Conference on Computer Languages, IEEE, San Francisco, CA, 1992.
18. Fuks, O., Ostroff, J.S., Paige, R.: *SECG: the SCOOP to Eiffel code generator*, in Journal of Object Technology, Volume 3, Number 10, 2004, pp. 143-160.
19. Owicki, S., Gries, D.: *Verifying properties of parallel programs: an axiomatic approach*, in Communications of the ACM, Volume 19, Number 5, May 1976, pp. 279-285.
20. Jones, C. B.: *Development Methods for Computer Programs including a Notion of Interference*, PhD thesis, Oxford University, June 1981.
21. Misra, J., Chandy, K.M.: *Proofs of networks of processes*. in IEEE Transactions of Software Engineering, Volume 7, Number 4, pp. 417-426, July 1981.
22. Jones, C.B.: *Wanted: a compositional approach to concurrency*, in *Programming methodology*, chapter 1, pp. 1-15, Springer-Verlag New York, 2003.
23. Nienaltowski P.: *Efficient Data Race and Deadlock Prevention in Concurrent Object-Oriented Programs*, OOPSLA'04 Doctoral Symposium, October 2004, Vancouver, Canada.
24. Arslan, V., Eugster, P., Nienaltowski, P., Vaucouleur, S.: *SCOOP: concurrency made easy*, in Meyer, B., Schiper, A., Kohlas, J. (Eds.) Dependable Systems: Software, Computing, Networks, 2006 (to appear).

Appendix

```
class PRODUCER

create
   make

feature {NONE} -- Creation
  make (a_buffer: separate BOUNDED_QUEUE [INTEGER])
         -- Creation procedure.
       do
         buffer := a_buffer
       ensure
         buffer = a_buffer
       end

feature -- Basic operations
   store (a_buffer: separate BOUNDED_QUEUE [INTEGER]; i: INTEGER)
         -- Store `i' in 'a_buffer'.
        require
         not a_buffer.is_full
       do
         a_buffer.put (i)
       ensure
         a_buffer.count = old a_buffer.count + 1
       end

   produce
         -- Produce elements and store them in `buffer'.
       do
         from
         until False
         loop
           store (buffer, 10)
         end
       end

   buffer: separate BOUNDED_QUEUE [INTEGER]
         -- Shared buffer.
end
```

Figure 6. Producer.

```
class CONSUMER

create
   make

feature {NONE} -- Creation
   make (a_buffer: separate BOUNDED_QUEUE [INTEGER])
         -- Creation procedure.
      do
         buffer := a_buffer
      ensure
         buffer = a_buffer
      end

feature -- Basic operations
   retrieved (a_buffer: separate BOUNDED_QUEUE [INTEGER]): INTEGER
         -- Element retrieved from ‘a_buffer’.
       require
         not a_buffer.is_empty
      do
         Result := a_buffer.item
         a_buffer.remove
      ensure
         a_buffer.count = old a_buffer.count - 1
      end

   consume
         -- Consume elements from `buffer’.
      local
         i: INTEGER
      do
         from
         until False
         loop
            i := retrieved (buffer)
         end
      end

   buffer: separate BOUNDED_QUEUE [INTEGER]
         -- Shared buffer.
end
```

Figure 7. Consumer.