DOI 10.1007/s00165-007-0063-2
BCS © 2007
Formal Aspects of Computing

Formal Aspects of Computing

Contracts for concurrency

Piotr Nienaltowski[1], Bertrand Meyer[2] and Jonathan S. Ostroff[3]

[1] Praxis High Integrity Systems Limited, 20 Manvers Street, Bath BA1 1PX, UK
E-mail: piotr.nienaltowski@praxis-his.com
[2] ETH Zurich, Zurich, Switzerland
[3] York University, Toronto, Canada

Abstract. The SCOOP model extends the Eiffel programming language to provide support for concurrent programming. The model is based on the principles of Design by Contract. The semantics of contracts used in the original proposal (SCOOP_97) is not suitable for concurrent programming because it restricts parallelism and complicates reasoning about program correctness. This article outlines a new contract semantics which applies equally well in concurrent and sequential contexts and permits a flexible use of contracts for specifying the mutual rights and obligations of clients and suppliers while preserving the potential for parallelism. We argue that it is indeed a generalisation of the traditional correctness semantics. We also propose a proof technique for concurrent programs which supports proofs—similar to those for traditional non-concurrent programs—of partial correctness and loop termination in the presence of asynchrony.

Keywords: Concurrency; Object-oriented programming; Design by contract; SCOOP; Software verification; Safety and liveness properties; Partial correctness

1. Introduction

Design by Contract (DbC) permits enriching class interfaces with assertions expressing the mutual obligations of clients and suppliers [Mey92]. Routine preconditions specify the obligations on the routine client and the guarantee given to the routine supplier; routine postconditions express the obligation on the supplier and the guarantee given to the client. Class invariants express the correctness criteria of a given class; an instance of a class is consistent if and only if its invariant holds in every observable state. The modular design fostered by DbC reduces the complexity of software development: correctness considerations can be confined to the boundaries of components (classes) which can be proved and tested separately. Clients can rely on the interface of a supplier without the need to know its implementation details.

We define the correctness of a routine as follows.

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

The sequential proof technique [Mey97] follows this definition which can be captured more formally by the following proof rule:

$$\frac{\{INV \wedge Pre_r\}\ \ body_r\ \ \{INV \wedge Post_r\}}{\{Pre_r[\overline{a}/\overline{f}]\}\ \ x.r(\overline{a})\ \ \{Post_r[\overline{a}/\overline{f}]\}} \tag{2}$$

Correspondence and offprint requests to: P. Nienaltowski, E-mail: piotr.nienaltowski@praxis-his.com

This rule says that if a routine `r` is locally correct (according to Definition 1) then a call to `r` in a state that satisfies its precondition will terminate in a state that satisfies its postcondition. Clients simply need to ensure that the precondition holds before the call; they may assume the postcondition in return.

It is tempting to apply the same rule to reasoning about concurrent programs. Unfortunately, the standard correctness semantics of assertions breaks down in a concurrent setting. In particular, clients may be unable to satisfy preconditions that depend on the state of shared objects; no matter how hard they try to establish such preconditions, other clients' actions may invalidate them. To deal with this problem, the original model [Mey97], called "SCOOP_97" here, uses two different semantics for preconditions, depending on whether they involve separate calls (calls targeting objects accessible to several clients). Separate preconditions are *wait conditions*, i.e. a violated precondition causes the client to wait. Non-separate ones are *correctness conditions*, i.e. a non-satisfied precondition is a contract violation and results in an exception. This mitigates the problem but the clash between wait conditions and correctness conditions is a source of other problems; for example a call may deadlock if it uses a non-separate actual argument for a separate formal and the associated wait condition does not hold. Furthermore, SCOOP_97 does not exploit the potential of other assertions in a concurrent context: postconditions, loop assertions and `check` instructions simply keep their sequential semantics. Separate postconditions are particularly problematic: they cause waiting, thus minimise the potential for parallelism and increase the likelihood of deadlock. Since interesting properties of concurrent programs are expressed as separate assertions, and such assertions are completely excluded from proof rules, this limits formal reasoning about concurrent programs.

We propose to solve these problems by "lifting" the principles of DbC to the concurrent context, so that assertions capture the full contract between a client and a supplier, including synchronisation. This requires a new, uniform semantics of contracts applicable in both concurrent and sequential contexts, and a modular proof technique which supports reasoning about interesting properties of concurrent programs in style similar to the above proof technique. We want both the new semantics and the new proof technique to reduce in a straightforward manner to their traditional counterparts when no concurrency is involved.

The article is organised as follows. Section 2 gives a summary of the computational model. (We use the refined SCOOP model described in the first author's PhD thesis [Nie07b]; a description of the original SCOOP_97 can be found in [Mey97].) Sections 3–6 discuss the semantics of preconditions, postconditions, invariants, and loop assertions respectively. Section 7 introduces the proof rule for asynchronous and synchronous calls. Section 8 points out the limitations of the proposed proof technique, and analyses the interplay of the new semantics with other SCOOP mechanisms. Section 9 discusses related work.

2. SCOOP model

Concurrency in SCOOP relies on the basic mechanism of object-oriented computation: the feature call[1]. Each object is handled by a *processor*—a conceptual thread of control—called the object's *handler*. All features of a given object are executed by its handler; as a result, only one processor may access any object at any time. Several objects may have the same handler; the mapping between an object and its handler does not change over time. If a client object and its supplier have the same handler, the feature call is synchronous; if they have different handlers, the call becomes asynchronous, i.e. the computation on the client's handler may move ahead without waiting. Objects handled by different processors are called separate from each other; objects handled by the same processor are non-separate. Likewise, we talk about separate and non-separate calls, entities, and expressions. A processor, together with the objects 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: 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*: illegal interleavings of calls from different clients. 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, meaning (as we will see in Definition 4 below) that the supplier is *controlled* by the client. Locking is achieved through a revised form of the mechanism of *feature application*: the processor

[1] A *feature* is either a routine or an attribute defined in a class.

```
store (buffer: separate BUFFER [INTEGER]; i: INTEGER)
        -- Store i in buffer.
    require
        not buffer.is_full
        i > 0
    do
        buffer.put (i)
    end

s_buffer: separate BUFFER [INTEGER]
ns_buffer: BUFFER [INTEGER]
...
store (s_buffer, 24)
store (ns_buffer, 10)
s_buffer := ns_buffer
store (s_buffer, 79)
```

Fig. 1. Preconditions in a concurrent context

executing a routine with attached formal arguments blocks until the processors handling these arguments have been locked (atomically) for its exclusive use; the routine serves as a critical section. 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. Clients resynchronise with their suppliers if and when necessary, thanks to *wait by necessity* [Car93]: clients wait only on queries (function or attribute calls) because they return some result; commands (procedure calls) do not require waiting.

3. Preconditions

In a sequential context, a precondition is a correctness condition: the client has to guarantee that the precondition holds at the moment of the call. The supplier expects the precondition to hold when the execution of the routine starts. Since all calls are synchronous, *feature application* (the action of applying a feature to a target object, performed by the target object's processor) follows the *feature call* (the action of calling a feature, performed by the client's processor); no other event may happen in between. Therefore, any property that holds at the call time also holds at the application time; conversely, any property that does not hold at the call time does not hold at the application time. Hence the usual understanding of the client's obligation: the precondition must hold at the call time.

In a concurrent context, the feature call and the feature application do not necessarily coincide; other events may happen in between. Therefore, a supplier cannot assume that a property satisfied at the call time still holds at the application time. (This is the source of the *separate precondition paradox* [Mey97].) Conversely, a property that does not hold at the time of the call may become true later on; the supplier could safely execute the feature at that time. These observations lead to treating preconditions as a synchronisation mechanism: a called feature cannot be executed unless the precondition holds; a violated precondition delays the execution of the feature. This *wait semantics* is reflected by the refined feature application mechanism and the feature application rule introduced in Sect. 6.1 of [Nie07b].

SCOOP_97 uses both semantics, depending on whether a given precondition clause involves separate calls. The precondition clause `i > 0` in Fig. 1 is a correctness condition because it involves no separate calls, whereas `not buffer.is_full` has the wait semantics. This distinction is somewhat artificial because `buffer`, although declared as `separate`, may denote a non-separate object at run time, for a call such as `store (ns_buffer, 10)`. Should wait semantics apply here? If yes, the client will be blocked forever if `ns_buffer` is full. In this example, the static type of the actual argument is non-separate; but the same problem may occur even if the actual argument has a separate type, for example in the subsequent call `store (s_buffer, 79)`. Clearly, wait semantics should be replaced by correctness semantics here.

Sometimes, the opposite is required: a correctness condition should be turned into a wait condition. This happens in the presence of polymorphism and feature redefinition. A class that redefines a feature may change the type of a formal argument from non-separate to separate; such redefinitions are safe because clients of the original class may only use a non-separate actual argument. The precondition clauses that involve the redefined

```
-- in class X
ns_buffer: BUFFER [INTEGER]
store (buffer: separate BUFFER [INTEGER]; i: INTEGER)
        -- Store i in buffer.
    require
        buffer_not_full: not buffer.is_full
        i_positive: i > 0
    do
        buffer.put (i)
    end

-- in class C
s_buffer: separate BUFFER [INTEGER]
ns_buffer: BUFFER [INTEGER]
r (x: separate X; buf: separate BUFFER [INTEGER])
    do
        x.store (s_buffer, 10) -- buffer_not_full uncontrolled
        x.store (ns_buffer, 24) -- buffer_not_full controlled
        x.store (buf, 79) -- buffer_not_full controlled
        x.store (x.ns_buffer, 19) -- buffer_not_full controlled
    end
```

Fig. 2. Controlled and uncontrolled preconditions clauses

formal argument are correctness conditions in the original feature but they become wait conditions in the redefined version.

The necessity to turn wait conditions occasionally into correctness conditions, and occasionally to do the reverse, suggests a uniform semantics of preconditions. We adopt the semantics captured by Definition 3.

Definition 3 (Semantics of preconditions)**.** A precondition expresses the necessary requirements for a correct application of a feature. The execution of the feature's body is delayed until the precondition is satisfied.

The guarantee given to the supplier is exactly the same as with the traditional semantics: the precondition holds on entry to the feature's body. Nevertheless, the definition does not force the clients to ensure the precondition before the call. A client performing a feature call naturally has certain obligations, but it must share the responsibility for establishing the precondition with the environment: the client should be blamed for any contract breaches that it could have avoided; the environment is responsible for everything that is not under the client's control. Meyer [Mey97] argues that only non-separate precondition clauses are binding for the client. We go one step further: since the client has full control over *controlled* expressions, which represent objects whose processors are locked by the client's processor (see Definition 4), it should be blamed for breaches in precondition clauses involving such expressions. Therefore, we extend the client's responsibilities to all *controlled* clauses.

Definition 4 (Controlled expression)**.** An expression `exp` is controlled if and only if `exp` is attached (statically known to be non-void) and either `exp` is non-separate or `exp` is handled by the same processor as some formal argument of the routine `r` in which `exp` appears.

Definition 5 (Controlled clause)**.** For a client performing the call $x.f(\overline{a})$ 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 $\overline{a}$ for the formal arguments, it only involves calls on expressions that are controlled in the context of `r`; otherwise, it is *uncontrolled*.

We have chosen assertion clauses as units of controllability, although finer-grained units could be used instead. Assertion clauses are convenient because they are clearly demarcated in the program text. Also, Eiffel follows this style for the representation of inherited and immediate assertions in redefined features.

Figure 2 illustrates the difference between controlled and uncontrolled precondition clauses. For the client executing the call `x.store (s_buffer, 10)` in the body of `r`, the first precondition clause of `store` is uncontrolled because, after the substitution of actuals for formals, it involves the expression `s_buffer.is_full` whose target `s_buffer` is not controlled in the context of `r`. On the other hand, the same precondition clause is controlled in the three remaining calls to `store` because the targets of the involved expressions are controlled in `r`: `ns_buffer` is non-separate; `buf` is separate but locked by `r`; `x.ns_buffer` is separate, but it is non-separate from `x` and `x` is controlled, therefore `x.ns_buffer` is controlled too (see Definition 4). The second precondition clause `i > 0` is controlled in all cases because the expected actual argument is handled by the same processor as the target `x`. (`x`

itself is controlled; otherwise, no calls on `x` would be allowed in `r`.) In this particular case, the actual argument is expanded[2], hence controlled in any context.

Coming back to the client's responsibilities: each controlled precondition clause must be satisfied by the client. But the calls on `x` in Fig. 2 are asynchronous; how can the client ensure a property if the previously scheduled calls are still being executed and may change the state of the involved objects? Indeed, the state of `x` and `buf` at the moment of the call `x.store (buf, 79)` may be different from their state at the moment the feature gets actually applied; the client knows, however, that all its calls on `x` and `buf` are performed in the FIFO order, allowing the client programmer to reason sequentially about the execution. That is, the state of `x` and `buf` at the moment of the application of `store (buf, 79)` is exactly the same as it was when `store (ns_buffer, 24)` terminated. The client only needs to ensure that the properties established by `x.store (ns_buffer, 24)` imply the precondition of `x.store (buf, 79)`. It does not matter that the precondition may not hold at the moment of the call; it will hold when required, at the moment of the feature application.

The establishment of uncontrolled precondition clauses depends on the emergent behaviour of the whole system. The client cannot take responsibility for such preconditions: calling a feature with an uncontrolled precondition clause may result in indefinite waiting; see Sect. 8.1 for a detailed discussion of this problem.

Although we use the same name for the proposed *wait semantics* of preconditions, there are three major differences with respect to SCOOP_97:

- The semantics applies to all preconditions; there is no distinction between separate and non-separate clauses.
- Clients are responsible for establishing all controlled precondition clauses, not only the non-separate ones.
- Waiting happens on the supplier side. For example, the client executing the call `x.store (s_buffer, 10)` in Fig. 2 is not blocked; it is `x`'s handler that waits until the precondition is satisfied. (If the client and the supplier are non-separate from each other, then waiting on the supplier side is equivalent to waiting on the client side.)

Even though a precondition violation causes waiting, this waiting is only conceptual in the case of controlled clauses: if a controlled precondition clause is violated when it is supposed to hold (at the moment of the feature application), waiting is useless because the state of the involved objects cannot change in the meantime as they are handled by the client's processor; an exception is raised instead. In a sequential setting, this semantics is equivalent to the traditional correctness semantics because all precondition clauses involve only non-separate expressions and are therefore controlled (see Definition 5). For example, the call `store (ns_buffer, 10)` in Fig. 1 can be handled correctly: if the precondition `not buffer.is_full` does not hold during the execution, the result will be to trigger an exception. Similarly, the application of `store` with a negative second argument results in an exception because the precondition `i > 0` is violated. Since the feature application happens immediately after the corresponding feature call, it looks as if the exception occurs at the moment of the call, just as with traditional sequential semantics.

4. Postconditions

The original design of SCOOP_97 applied the standard correctness semantics to separate postconditions; they are evaluated synchronously and understood to hold immediately when the feature call terminates. Meyer [Mey97] discusses the *separate postcondition paradox*: on return from a separate call, the client cannot be sure that the postcondition still holds: the processor handling the involved supplier may become free in the meantime so that other clients may modify its state, thus invalidating the postcondition. Rodriguez et al. [RDF+05] refer to the same problem as *external interference*.

The treatment of postconditions in SCOOP_97 is unsatisfactory for two reasons:

- Separate postconditions are excluded from the proof rule for feature calls; this limits considerably the range of safety properties that can be proven.
- The synchronous evaluation of separate postconditions limits the parallelism and may lead to deadlocks.

Suppose the client executing the call `spawn_two_activities (york, tokyo)` in Fig. 3 wants to launch jobs at two independent locations, with the guarantee that the job at each location will eventually terminate. Such guarantees are most naturally expressed as postconditions of `spawn_two_activities`. The synchronous semantics

[2] An expanded target represents an object rather than a reference to an object; expanded targets are always non-separate.

```
spawn_two_activities (location_1, location_2: separate LOCATION)
    do
        location_1.do_job
        location_2.do_job
    ensure
        post_1: location_1.is_ready
        post_2: location_2.is_ready
    end

tokyo: separate LOCATION

r (york: separate LOCATION)
    do
        spawn_two_activities (york, tokyo)
        do_local_stuff
        get_result (york)
        do_local_stuff
        get_result (tokyo)
    end

get_result (location: separate LOCATION)
    require
        pre_1: location.is_ready
    do
        ...
    end
```

Fig. 3. Postconditions in a concurrent context

of postconditions would block the client until the call terminates: the subsequent operation `do_local_stuff` would not be performed until both postcondition clauses `post_1` and `post_2` of `spawn_two_activities` have been evaluated. The amount of effective parallelism would be much lower than without the postconditions, where the client is able to move on immediately (but there is no guarantee that the jobs are done). To take advantage of parallelism without sacrificing the guarantees given to the client, postconditions should be evaluated *asynchronously*, that is to say treated similarly to command calls: wait by necessity should not apply. As a result, the client can now move on with its next activity (`do_local_stuff`) without waiting for the evaluation of `post_1` and `post_2`; but it still gets the guarantee that both clauses will be satisfied *eventually*. More precisely, they will be satisfied when the execution of the features called within the body of `spawn_two_activities` terminates. Because `york` and `tokyo` are separate from the client (and from each other), the calls to `do_job` are asynchronous, so `post_1` and `post_2` do not necessarily hold when the client executes `do_local_stuff`; but this does not matter because `york` and `tokyo` are not used in that operation. The clause `post_1` becomes relevant only at the moment of the call to `get_result (york)`, which cannot start its execution until all the previous calls on `york` have terminated. Therefore, `get_result` cannot be executed until `york` terminates the job requested by `spawn_two_activities`; when this happens, the postcondition `post_1` stating that `york.is_ready` holds. This property implies the precondition of the routine `get_result`, which as a result can now be executed. The client is not penalised by the asynchronous evaluation of postconditions; it gets the same guarantees as with the synchronous semantics but projected into the future (a postcondition may be assumed to hold immediately after the execution of a feature's body although it may not hold at that point).

This semantics can be further refined to allow independent evaluation of postcondition clauses. In the above example, the state of `tokyo` is irrelevant for `get_result (york)`; it only becomes important later on, when `get_result (tokyo)` is called. The client is only interested in the first postcondition clause involving `york`; if this clause holds, `get_result (york)` proceeds without waiting for `tokyo`. This enables more parallelism, in particular if `tokyo` is much slower than `york`. Once again, the client is not penalised because it gets all the guarantees in due time. Definition 6 captures the new semantics of postconditions.

Definition 6 (Semantics of postconditions)**.** A postcondition describes the result of a feature's application. Postconditions are evaluated asynchronously. Postcondition clauses that do not involve calls on targets handled by the same processors are evaluated independently.

Following Definition 5, in the call `spawn_two_activities (york, tokyo)`, the postcondition clause `post_1` is controlled whereas `post_2` is uncontrolled. The client may assume the former (`york.is_ready`) after the call because no other client can invalidate it; the latter clause (`tokyo.is_ready`) cannot be assumed because of the possible interference by other clients. The situation is symmetric to the treatment of preconditions: a controlled

```
-- in class X
r (a: A)
    do
        ...
    ensure
        post_1: a.q
        post_2: p (a)
    end

-- in class C
my_x: X
my_a: A
...
my_x.r (my_a)
do_local_stuff
```

Fig. 4. New postcondition semantics in a sequential context

precondition clause constitutes an obligation on the client; a controlled postcondition clause is a guarantee given to the client. We use this observation to derive a proof rule for feature calls in Sect. 7.

Each query call appearing in a postcondition clause is evaluated by its target's handler; the results of all the queries are combined into one boolean value (This operation is performed by a processor involved in the given clause. The implementation of SCOOP described in [Nie07b] picks the first processor in the order of appearance). The violation of a postcondition clause raises an exception in the processor that has executed the routine. This gives rise to the problem of *asynchronous exceptions*: the client's processor might have already left the context of the call, so the exception cannot be propagated correctly. This topic is beyond the scope of this article; see the exception handling mechanisms proposed by Arslan and Meyer [AM06], Brooke and Paige [BP07], and Adrian [Adr02].

We may observe how the semantics of postconditions reduces to the traditional sequential semantics if no concurrency is involved. Consider the call `my_x.r (my_a)` in Fig. 4. The client, the supplier `my_x`, and the actual argument `my_a` are handled by the same processor. The client cannot proceed to `do_local_stuff` until both postcondition clauses have been evaluated because the client's processor is in charge of evaluating them. As a result, either all the postcondition clauses hold immediately after the call or an exception is raised. Also, the client can assume all the postconditions of `r` because they are controlled. This corresponds to the sequential semantics.

5. Invariants

Invariants play an important role in the DbC methodology. They are the primary tool for ensuring the consistency of objects: an instance of class `C` is consistent if and only if it satisfies the invariant of `C`. The invariant must be satisfied in every observable state, i.e. in any state where the object may be accessed by a client.

The standard Eiffel semantics applies to class invariants in SCOOP because all the expressions appearing in invariants must be non-separate. There is no explicit rule prescribing it; however, the controllability requirement imposed by the new call validity rule [Nie07b, rule 6.5.3] can only be satisfied by non-separate entities because invariants have no enclosing routines. The invariant is checked before and after the application of a feature (unless the feature has been called using an unqualified form, i.e. with the `Current` object as the implicit target); an invariant violation raises an exception in the supplier.

6. Loop assertions and check instructions

We apply the asynchronous semantics to loop variants, loop invariants, and `check` instructions.[3] Wait by necessity does not apply: the evaluation of an assertion is blocking only if the current processor is involved, i.e. the assertion includes non-separate expressions. All loop variants, invariants, and `check` instructions are *controlled* because all the involved entities serving as targets of calls must be controlled; the call validity rule would be violated otherwise. Therefore, such assertions preserve their contractual character and may be used for formal reasoning.

[3] Check instructions are similar to `assert` statements in Spec♯ [BLS04] and JML/Java [LPC+05].

```
remove_n (list: separate LIST [G]; n: INTEGER)
        -- Remove n elements from list.
    require
        list.count >= n
    local
        initial, removed: INTEGER
    do
        from
            initial := list.count
            removed := 0
        until
            removed = n
        invariant
            list.count + removed = initial
        variant
            list.count - initial + n -- the same as n-removed
        do
            list.remove
            removed := removed + 1
        end
    ensure
        list.count = old list.count - n
    end
```

Fig. 5. Loop assertions in a concurrent context

Similarly to postconditions, their asynchronous evaluation does not influence the reasoning style: they may be assumed to hold immediately. When a loop assertion or a check evaluates to `False`, an exception is raised.

Figure 5 illustrates the advantages brought by the asynchronous semantics. The assertions appearing in the body of `remove_n` capture the essence of the loop: at each iteration, the number of elements in `list` is reduced, and the number of remaining elements plus the number of removed elements equals the initial number of elements. If the variant and invariant are to be evaluated, the evaluation will be asynchronous; the processor handling `list` is asked to evaluate them in due time, after the previous features targeting `list` but before the next application of `remove`. The request queue of `list`'s handler may look like this:

$$P_{list} : ...[list.remove][eval_{var}][eval_{inv}][list.remove][eval_{var}][eval_{inv}][list.remove]...$$

The client does not wait; it performs the next iteration of the loop, until the exit condition becomes true. Even if the handler of `list` is very slow, so that the application of `remove` or the evaluation of assertions takes a long time, the client may forsake waiting for the execution of `remove_n` to terminate, and still rely on the guarantee that n elements will eventually be removed. (If the call to `remove_n` occurs in a context where the actual argument is controlled then the postcondition is controlled and the client may assume it immediately after the call.) Even though the assertions involve separate calls, the correctness and the termination of the loop, as well as the establishment of the postcondition can be proved using the standard sequential reasoning. The new semantics maximises the amount of parallelism while preserving the sequential reasoning style.

Check instructions are treated in the same way as loop assertions; they are evaluated asynchronously in due time. For example, the following sequence of calls:

x.f; x.set_i (10); **check** x.i = 10 **end**; x.g

results in checking `x.i = 10` after the application of `x.set_i (10)` but before the application of `x.g`. The request queue of `x`'s handler looks like that:

$$P_x : ...[x.f][x.set\_i(10)][eval_{x.i=10}][x.g]...$$

The client is sure that `x.i = 10` is true before the *application* of `x.g`. Note the difference between this guarantee and the guarantee given by an explicit `if` instruction

if x.i = 10 **then** x.g **else** ... **end**

Here, due to wait by necessity, the client is sure that `x.i = 10` holds before *calling* `x.g`. This is a stronger guarantee but it comes at the cost of reduced parallelism.

7. Towards a proof rule

The semantics of preconditions discussed in Sect. 3 implies that a client's responsibilities should be limited to establishing the controlled precondition clauses before the call; the uncontrolled ones may only be established by the environment, through the emergent behaviour of other processors. Similarly, the semantics of postconditions introduced in Sect. 4 lets clients assume controlled postcondition clauses after the call; the uncontrolled ones hold when the routine terminates but the client cannot assume them, due to the potential interference of other clients. We use this observation to define the mutual obligations of clients and suppliers, thus establishing the contractual character of assertions and capturing the essence of DbC in a general (potentially concurrent) context.

Definition 7 (Mutual obligations of clients and suppliers)**.** The supplier may assume all the precondition clauses at the entry to the feature's body; it must establish all postcondition clauses when the body terminates. The client must satisfy all the controlled precondition clauses at the moment of the call; it may assume all the controlled postcondition clauses after the call.

The above definition applies to synchronous and asynchronous calls but needs no temporal operators: sequential reasoning applies here. This is possible because controlled assertions may be projected into the future, i.e. assumed to hold immediately even if their evaluation is delayed. From the supplier's point of view, all the preconditions and postconditions are controlled; therefore, its obligations are the same as in a sequential context. The client, however, has fewer obligations; conversely, it gets fewer guarantees. The Hoare-style Rule 8 derived from the sequential proof rule for feature calls formalises Definition 7. 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; therefore, all of them appear in the antecedent. The invariant must be fully controlled to avoid side effects: the call validity rule prohibits separate expressions, but expressions of the form `q (x)` must also be prohibited if `x` is separate; otherwise, the evaluation of *INV* might be blocking and subsequent evaluations might yield different results. The proof rule now becomes:

$$\frac{\{INV \wedge Pre_r\}\ \ body_r\ \ \{INV \wedge Post_r\}}{\{Pre_r^{ctr}[\overline{a}/\overline{f}]\}\ \ x.r(\overline{a})\ \ \{Post_r^{ctr}[\overline{a}/\overline{f}]\}} \tag{8}$$

The following sequential-like proof technique for partial correctness of synchronous and asynchronous feature calls relies on the newly introduced proof rule.

Definition 9 (Proof technique for synchronous and asynchronous calls)**.** Consider a call $x.r(\overline{a})$. If we can prove that the body of `r`, started in a state satisfying the precondition, satisfies the postcondition if and when it terminates, then we can deduce the same property for the above call, with actual arguments $\overline{a}$ substituted for the corresponding formal arguments, every non-qualified call in the assertions (of the form *some_property*) replaced by the corresponding property on `x` (of the form *x.some_property*), and ignoring the uncontrolled assertions.

Our approach eliminates the need for a special treatment of asynchrony. It makes it possible to reason about sequences of asynchronous calls—or interleaved synchronous and asynchronous calls—without using temporal operators. For example, it is easy to demonstrate the correctness of the feature `store_three` in Fig. 6 even though its body contains a sequence of separate calls. The precondition of the first call to `buffer.put` is implied by the precondition of `store_three`. Its postcondition is assumed to hold immediately after the call; the postcondition implies the precondition of the second call to `buffer.put`. The postcondition of the second call — again assumed immediately after the call — implies the precondition of the third call to `buffer.put`. Finally, the postcondition of the third call implies the postcondition of `store_three`. This technique is modular: the proof of `store_three` relies on the contract of {`BUFFER`}`.put`; once we have proved the correctness of `store_three`, its contract can be used for proving the correctness of routines that call `store_three`, and so on.

8. Discussion

8.1. Limitations of the proof technique

Rule 8 is strong enough to prove partial correctness of programs; certain liveness properties, such as loop termination, can also be proved. Total correctness, however, cannot be proved because of the potential deadlocks

```
-- in class C
store_three (buffer: separate BUFFER [INTEGER]; i, j, k: INTEGER)
        -- Store i, j, and k in buffer.
    require
        buffer.count <= buffer.size − 3
    do
        -- {not buffer.is_full}
        buffer.put (i)
        -- {buffer.count = old buffer.count + 1}
        -- ===>
        -- {buffer.count = old buffer.count + 1 and not buffer.is_full}
        buffer.put (j)
        -- {buffer.count = old buffer.count + 2}
        -- ===>
        -- {buffer.count = old buffer.count + 2 and not buffer.is_full}
        buffer.put (k)
        -- {buffer.count = old buffer.count + 3}
    ensure
        buffer.count = old buffer.count + 3
    end

-- in class BUFFER [G]
put (v: G)
        -- Store v.
    require
        not is_full
    ensure
        count = old count + 1
```

Fig. 6. Assertional reasoning about asynchronous calls

and infinite waiting on non-satisfied uncontrolled preconditions. Recall the York–Tokyo example from Fig. 3. The calls `spawn_two_activities (york, tokyo)`, `get_result (york)`, and `get_result (tokyo)` happen in a context where `york` is controlled but `tokyo` is not. Therefore, all assertion clauses involving `tokyo` are uncontrolled; the proof rule ignores them. Assume that the feature `get_result` is defined as

```
get_result (location: separate LOCATION)
    require
        location.is_ready
    ensure
        location.result_retrieved
```

and the postcondition of `do_local_stuff` is empty. The proof sketch for the routine `r` in Fig. 7 demonstrates that reasoning about `york`'s properties is not hindered: the client has to establish the precondition of `get_result (york)` using the postcondition of `spawn_two_actitivies`; it can prove the postcondition of `r` using the postcondition of `get_result (york)`. The establishment of the precondition clauses involving `tokyo` is beyond the client's control: the client does not need to prove anything about `tokyo` but is not even sure whether the calls involving `tokyo` will ever proceed. The first possibility is that `tokyo`'s handler is never released by its current client; both calls `spawn_two_activities (york, tokyo)` and `get_result (tokyo)` may block for that reason. The second possibility is that `tokyo` becomes available but its state never satisfies the precondition (this may only happen for `get_result (tokyo)` because it has a non-trivial precondition). As a result, the total correctness of `r` cannot be proved without considering the behaviour of the whole system; but this requires global reasoning and the use of temporal operators to express such properties as "`tokyo` will eventually become available in a state satisfying `tokyo.is_ready`". Alternatively, Rely/Guarantee specifications [Jon81, MC81, Jon03] can be used to reason about the influence of the environment. Ostroff et al. [OTHS07] propose a technique to deal with properties of uncontrolled targets and introduce the concept of a *controlled routine* to define contexts in which our proof rule can be used to reason about total correctness of feature calls.

8.2. Contract redefinition

The proposed semantics of contracts and the proof technique are sound in the presence of polymorphism and dynamic binding; clients are not deceived even if contracts are redefined. The standard DbC rules apply: preconditions may be kept or weakened; postconditions and invariants may be kept or strengthened. A redefined precondition results in fewer obligations on the client and (potentially) less waiting at the moment of the feature

```
r (york: separate LOCATION)
    do
        -- {True}
        spawn_two_activities (york, tokyo)
        -- {york.is_ready and True}
        do_local_stuff
        -- {york.is_ready}
        -- ===>
        -- {york.is_ready}
        get_result (york)
        -- {york.result_retrieved}
        do_local_stuff
        -- {york.result_retrieved}
        -- ===>
        -- {york.result_retrieved and True}
        get_result (tokyo)
        -- {york.result_retrieved and True}
    ensure
        york.result_retrieved
    end
```

Fig. 7. Limitations of the proof technique

application; a redefined postcondition gives the client more guarantees. Chapters 7 and 10 of [Nie07b] discuss contract soundness in conjunction with the refined locking mechanism, and the support for polymorphism and dynamic binding in detail.

8.3. Importance of lock passing

The proof technique proposed here assumes the semantics of argument passing enriched with the *lock passing mechanism* introduced in [Nie07a, Nie07b]: if a feature call `x.f` (a_1, ..., a_n) occurs in the context of the routine `r` where some actual argument a_i is *controlled*, i.e. a_i is attached and locked by the processor executing `r`, and the corresponding formal argument of `f` is declared as attached, the client's handler (the processor executing `r`) passes all currently held locks (including the implicit lock on itself) to the handler of `x`, and waits until `f` has terminated; when the execution of `f` is complete, the client's handler resumes the computation.

Indeed, the proof technique would not be sound without the above mechanism. Consider the proof sketch in Fig. 8. The call `x.transfer_to (y)` is processed synchronously because `y` is controlled; therefore, lock passing occurs at the moment of the call. Any calls on `y` within the body of `transfer_to` are guaranteed to be executed before the subsequent call to `y.empty` issued by the client. Therefore, the postcondition of `transfer_to` may be assumed before the call `y.empty`; it is necessary to prove the correctness of that call (and the whole routine `s`). If no lock passing occurred (as in SCOOP_97), the call `y.empty` would be processed before the calls issued by the body of `transfer_to`; the assumption `y.is_full` would be false and the call `y.empty` invalidated. In fact, following SCOOP_97 rules, `x` would be able to execute `transfer_to` only after the client terminated `s` and unlocked `y`; therefore, `y.is_full` would eventually become true, which contradicts the promise made by `s`: its postcondition says that `y` is empty. See [Nie07b] for a more detailed discussion of lock passing and related issues.

8.4. Run-time assertion checking

Meyer [Mey97] mentions the problem of run-time assertion checking in a concurrent context, concluding: "*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.*"

In fact, assertion monitoring may be turned off even for wait conditions. Since the client is responsible for establishing the controlled precondition clauses, the runtime checks of such preconditions are not necessary, provided that the calls have been proved correct (or, at least, we are convinced that they are correct). Uncontrolled precondition clauses must be monitored; but even here some optimisations are possible. If an uncontrolled

```
-- in class C
s (x, y: separate X)
    require
        x.is_empty and y.is_empty
    do
        -- {x.is_empty and y.is_empty}
        x.fill
        -- {x.is_full and y.is_empty}
        x.transfer_to (y)
        -- {y.is_full}
        y.empty
        -- {y.is_empty}
    ensure
        y.is_empty
    end

-- in class X
fill
        -- Fill container.
    require
        is_empty
    ensure
        is_full

empty
        -- Remove all elements.
    require
        is_full
    ensure
        is_empty

transfer_to (y: separate X)
        -- Transfer contents to y.
    require
        y.is_empty
    ensure
        y.is_full
```

Fig. 8. Interplay of lock passing and contracts

precondition clause holds whenever the feature is applied—for example the clause follows from the class invariant, or maybe it requires some (monotonic) property that has already been established by an earlier call—then there is no point in re-checking the clause every time; we know that it holds.

This reveals another advantage of the new semantics of postconditions, checks, and loop assertions: the run-time monitoring of assertions has no impact on the semantics of correct programs (except for the incurred overhead). With the traditional sequential semantics, wait by necessity forces the client to wait for the evaluation of an assertion if the run-time checking is on but no waiting happens when it is turned off; the program's behaviour changes. Our semantics eliminates such inconsistencies.

9. Related work

Ostroff et al. [OTHS07] show that contracts provide only a certain measure of correctness but do not guarantee additional safety and liveness properties without global reasoning. They model SCOOP programs as fair transition systems [MP95] and use temporal logic for describing and proving system properties beyond contractual correctness. This approach is complementary to ours: it permits expressing and proving properties of uncontrolled targets which cannot be captured by our proof rules.

Bailly [Bai04] 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 the traditional (correctness) preconditions. The approach does not support inheritance, therefore it does not address problems caused by guard strengthening vs. precondition weakening. The treatment of postconditions is identical as in the sequential context, although the author comments on the infeasibility of formal reasoning

with the sequential proof rule. Other concurrent extensions of Eiffel, such as CEiffel [Löh92], CEE [Jal94], and Distributed Eiffel [GL92] rely on guard-based synchronisation with a syntactic distinction between guards and preconditions. The distinction between guards and preconditions has deep implications for formal reasoning: clients need to satisfy preconditions but not guards; in the context of polymorphism, guards can be strengthened whereas preconditions can only be kept or weakened.

Sutton [Sut95] describes a new strategy for condition-based process execution, based on delayed evaluation of preconditions and postconditions. Preconditions have guard semantics but they are evaluated in parallel with task (routine) bodies; a task might be allowed to execute even if some of its preconditions have not yet been evaluated. They only have to hold at a particular point of execution; otherwise, the task is put on hold or cancelled. The task may terminate even if some of its postconditions have not been established. They have to be established eventually; otherwise, the task must be cancelled (rolled back or compensated). This semantics is similar to ours: postconditions are also projected into the future. The precondition semantics proposed by Sutton may be simulated in our model by splitting up a routine into smaller subroutines which require their preconditions to hold on entry to their bodies.

Rodriguez et al. [RDF+05] propose a concurrent extension of JML whereby method guards are treated in a similar way to Sutton's preconditions. Guards are specified using the keyword `when` and appear in a feature header, after the preconditions. If a feature is called in a state where its guard does not hold, it should block until the guard is satisfied. Interestingly, 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 assumed to be at the end of the body. There is no implicit waiting mechanism; programmers must write synchronisation code by hand, for example using busy-waiting loops. The approach facilitates static verification of atomicity properties but it does not simplify the construction of concurrent programs.

10. Conclusions

We have outlined a new semantic approach to contracts—routine preconditions and postconditions, loop variants and invariants, and `check` instructions—which lifts the principles of Design by Contract from purely sequential programming to the concurrent context, while remaining applicable to sequential object-oriented programs. Preconditions become wait conditions; separate postcondition clauses earn an asynchronous semantics and are evaluated independently from other postcondition clauses. Similarly, loop assertions and check instructions are evaluated asynchronously. The resulting programming model makes it possible to exploit the potential for parallelism while preserving the intuitive meaning of contracts as a way to express the mutual obligations of clients and suppliers. We have argued that the proposed semantics is indeed a generalisation of the traditional sequential semantics: they are equivalent when no concurrency is involved. The contract semantics presented here is supported by the current implementation of SCOOP, available for download at *http://se.ethz.ch/research/scoop.html*.

We have also proposed a proof technique for reasoning about safety properties of object-oriented concurrent programs. It builds upon the new contract semantics and the Hoare-style rule for feature calls which permits guarantees expressed in assertions to be projected into the future. As a result, it becomes possible to treat synchronous and asynchronous calls in a similar way; this greatly reduces the complexity of reasoning formally about concurrent code.

Our treatment of contracts and the proof technique are compatible with such essential object-oriented mechanisms as inheritance, polymorphism, and dynamic binding.

Acknowledgments

We would like to thank the FACJ referees and CORDIE'06 participants for their comments on earlier versions of this article. We are also grateful to Peter Müller for important feedback on the proof technique as well as to Volkan Arslan and many ETH students for numerous comments and discussions on the model and its implementation. This research work was conducted during the first author's PhD studies at the ETH Zurich. We are grateful for the generous support of the Hasler Foundation as part of its DICS project on Dependable Information and Communication Systems [AENV06] (DICS project number 1834), and of the Swiss National Science Foundation (FNS project number 200021-100498/1).

References

[Adr02] Adrian C (2002) SCOOP for SmallEiffel. draft, available online at http://www.chez.com/cadrian/eiffel/scoop.html, June 2002

[AENV06] Arslan V, Eugster P, Nienaltowski P, Vaucouleur S (2006) SCOOP: concurrency made easy. In: Meyer B, Schiper A, Kohlas J (eds) Dependable systems: software, computing, networks. Springer, Heidelberg

[AM06] Arslan V, Meyer B (2006) Asynchronous exceptions in concurrent object-oriented programming. In: International Symposium on Concurrency, Real-Time, and Distribution in Eiffel-like Languages (CORDIE), York, UK, July 2006

[Bai04] Bailly A (2004) Formal semantics and proof system for SCOOP. White paper, October 2004

[BLS04] Barnett M, Leino KRM, Schulte W (2004) The Spec♯ programming system: an overview. In: CASSIS, vol 3362 of LNCS. Springer, Heidelberg

[BP07] Brooke PJ, Paige RF (2007) Exceptions in Concurrent Eiffel. J Object Technol 6(10):111–126

[Car93] Caromel D (1993) Towards a method of object-oriented concurrent programming. Commun ACM 36(9):90–102

[GL92] Gunaseelan L, LeBlanc RJ (1992) Distributed eiffel: a language for programming multigranular objects. In: 4th International conference on computer languages, San Francisco

[Jal94] Jalloul G (1994) Concurrent object-oriented systems: a disciplined approach. PhD thesis, University of Technology, Sydney

[Jon81] Jones CB (1981) Development methods for computer programs including a notion of interference. PhD thesis, Oxford University

[Jon03] Jones CB (2003) Wanted: a compositional approach to concurrency, Chapter 1. Springer, Heidelberg, pp 1–15

[Löh92] Löhr K-P (1992) Concurrency annotations. ACM SIGPLAN Notices 27(10):327–340

[LPC+05] Leavens GT, Poll E, Clifton C, Cheon Y, Ruby v, Cok DR, Kiniry J (2005) JML reference manual. Iowa State University, Department of Computer Science

[MC81] Misra J, Chandy KM (1981) Proofs of networks of processes. IEEE Trans Softw Eng 7(4):417–426

[Mey92] Meyer B (1992) Applying "Design by contract". IEEE Comput 25(10):40–51

[Mey97] Meyer B (1997) Object-oriented software construction, 2nd edn. Prentice Hall, Englewood Cliffs

[MP95] Manna Z, Pnueli A (1995) Temporal verification of reactive systems: safety. Springer, New York

[Nie07a] Nienaltowski P (2007) Flexible access control policy for SCOOP. Formal Aspects of Computing, special issue: Concurrency, Real-Time, and Distribution in Eiffel-like Languages (CORDIE), (to appear)

[Nie07b] Nienaltowski P (2007) Practical framework for contract-based concurrent object-oriented programming. PhD thesis, no. 17061, Department of Computer Science, ETH Zurich

[OTHS07] Ostroff J, Torshizi FA, Huang HF, Schoeller B (2007) Beyond contracts for concurrency. Formal Aspects of Computing, special issue: Concurrency, Real-Time, and Distribution in Eiffel-like Languages (CORDIE), (to appear)

[RDF+05] Rodriguez E, Dwyer M, Flanagan C, Hatcliff J, Leavens GT, Robby (2005) Extending JML for modular specification and verification of multi-threaded programs. In: European Conference on Object-Oriented Programming (ECOOP), pp 551–576

[Sut95] Sutton SM (1995) Preconditions, postconditions, and provisional execution in software processes. Technical Report UM-CS-1995-077, University of Massachusetts, Amherst

Received 12 March 2007

Accepted in revised form 19 November 2007 by P. J. Brooke, R. F. Paige and Dong Jin Song