

Concurrent Contracts and Inter-Object Synchronization in MP-Eiffel

Miguel Oliveira e Silva

IEETA—DETI, Universidade de Aveiro, Aveiro, Portugal,
mos@det.ua.pt,
WWW home page: <http://www.ieeta.pt/~mos>

(This is draft work in progress article)

Abstract. This article presents part of a larger ongoing work which is related to the semantics of concurrent contracts and to the related problem of exclusive reservation of external objects. A prototype language – named MP-EIFFEL – in which all these mechanisms are being constructed and tested is also briefly presented, and compared with SCOOP’s concurrent approach.

1 Introduction

Design by ContractTM [2] is a methodology aiming to construct reliable programs. Contracts are expressed through assertions: boolean conditions that are required to be observed in the context in which they are declared.

There are several different types of assertions. The most important ones are class exportable assertions: routine’s preconditions and postconditions, and class invariants. These three types of assertions unambiguously express a contract in the usage of a class’s routine: preconditions are the conditions that clients are required to ensure in order to gain the benefit of the routine’s postcondition. Invariants express class wide conditions required to be observed in any (external) use of its services. A precondition failure is the responsibility of the routine caller. On the other hand, the responsibility of a failure of an invariant or a postcondition lies in the class itself.

In sequential programs the meaning and behavior of assertions¹ is very clear: a correct program does not depend on their existence and its behavior will be the same regardless of their runtime verification. However, if a program is not statically proved to be correct, there is the possibility for an assertion to be false at runtime. If a runnable assertion is false then the program is incorrect and an exception, signaling a broken contract, is immediately raised. Assertions and exceptions are closely related. The former express what to expect from a correct program. The latter gives the opportunity for a program to respond, when possible, to a runtime program error. Both aim to increase the reliability of software.

¹ Formal runnable assertions to be more precise.

In this article we study the problems related to concurrent contracts and to external object reservation. The behavior of concurrent contracts is narrowed to meaningful concurrent semantics, and their relationship with conditional synchronization and external object reservation (inter-object synchronization) is clarified.

All these proposals were recently adopted in MP-EIFFEL [5], which is an alternative proposal to extend EIFFEL with concurrent language constructs that is being developed by the author.

2 The SCOOP Approach

SCOOP [3, pages 951–1036] is a simple proposal to extend EIFFEL with concurrent language constructs. A unique new language keyword, **separate**, is enough for a simple, and yet quite rich, set of concurrent language mechanisms. The SCOOP model of concurrency uses direct inter-processor message-based communications. An invocation of a service belonging to a separate object (required to be attached, by the static type system, to a separate entity), is a message sent to the processor that created the object (called the object owner processor). The separate call rule [3, page 985] requires that only separate formal arguments can be used as a target for separate calls.

In SCOOP's original proposal (separate call semantics [3, page 996]), objects attached to formal separate arguments are required to be exclusively reserved during the entire execution of the routine. That execution, if necessary, will be postponed until such requirement is met. If the routine's precondition uses a separate formal argument, then not only will the routine have to wait for every separate object to be exclusively available, but those objects are required to verify the concurrent precondition.

Preconditions involving separate formal arguments are named concurrent preconditions in SCOOP, and their behavior is similar to conditional synchronization points: the processor will wait and reserve separate objects only when the concurrent preconditions hold.

At first sight, this behavior might seem to depart from the normal sequential semantics of preconditions. However, a closer look at the meaning of assertions gives a much clearer understanding of why this behavior is required. Assertions are correctness conditions. In the sequential world there is only one processor involved in the execution of a program, hence the responsibility of a false precondition can only belong to the processor attempting to execute the routine. So, no other behavior is acceptable than to signal a broken contract. In concurrent programs, on the contrary, preconditions may depend (indirectly, in SCOOP) on more than one processor, so most likely it won't be correct to state that the responsibility of a false concurrent precondition belongs to the processor which is evaluating that assertion. This situation was named by Meyer as the precondition paradox [3, page 994]. The only safe behavior, free from any race condition, for preconditions that depend on more than one processor, is to make them conditional synchronization points.

SCOOP's rule of restricting separate qualified calls to formal separate arguments together with its object reservation policy prevents the existence of concurrent postconditions (the object is required to be already exclusively reserved during the execution of the routine's body). Therefore any qualified call to a separate entity in a postcondition has the normal sequential semantics.

Since SCOOP forbids the existence of qualified calls to separate attributes, the invariant won't be able to use separate objects attached to attributes. There is, however, the possibility to use separate attributes indirectly through a boolean function with appropriate separate formal arguments (the invariant could call this function passing the separated attribute as the actual argument). So invariants can be used indirectly to represent concurrent assertions (but, unlike preconditions, without the guarantee that the concurrent condition would be verified when services begin its execution).

So far our SCOOP presentation has been based on its original proposal. However, recently [4], it is been proposed a (slightly) different approach to the problem of object reservation. Instead of always grabbing objects attached to separate formal arguments, that behavior is only required to occur at routine invocation in non-detachable separate types. Object reservation also occurs when a detached expression is proved to be attached (for example in a conditional instruction testing the non-voidness of an expression). Since a correct qualified call requires a non-void entity, and separate calls are always qualified calls, this approach is a simple and safe mechanism to reserve separate objects.

3 MP-Eiffel: A Brief Introduction

In MP-EIFFEL [5, 6] we are proposing a different approach to many of these issues. One of the most important differences is that, unlike SCOOP, MP-EIFFEL implements both types of inter-processor communication models: message passing and shared memory.

The inter-processor message passing mechanism requires the explicit use of a mechanism named *trigger*. The message sender is required to use a trigger calling instruction, which is syntactically similar to a normal service call, except that it uses the prefix keyword **trigger**. Triggerable services must also be explicitly declared as such in the receiver object's class. Trigger receive declarations (expressed as trigger clauses similar to creation clauses) are part of the public interface of classes. A class is free to declare any of its services to be a trigger. Like the creation clauses, the export status of triggers is independent of the exports declared in the feature clause in which the service exists (enabling the possibility for a service to be exported to different clients depending on whether they are being used as a normal or a triggered service).

The shared memory inter-processor model is implemented through the use of two new type qualifiers: **shared** and **remote**. A shared entity can be attached to shared objects. Shared object's services can be used by any processor with a reference to it. A remote entity can be attached to normal objects of other processors. Only query services without side-effect can be used through remote

entities. A call to a service of a shared or a remote object is required to not compromise its Abstract Data Type implementation. A sufficient condition, adopted in MP-EIFFEL, which achieves that goal is linearizability [1]. MP-EIFFEL allows the use of different intra-object synchronization schemes (monitor, exclusive readers-writer, concurrent readers-writer, lock-free) as long as it is possible, statically, to ensure linearizability in the object they protect [6].

Regardless of the intra-object synchronization scheme selected, it is also necessary to allow a correct inter-object synchronization scheme to grab concurrent objects for the exclusive use of external clients. The existence of both synchronization schemes within concurrent objects requires a solution allowing their peaceful and correct coexistence. A possible solution has been proposed in [6].

Originally [5], the proposed inter-object reservation language mechanism was similar to SCOOP's original proposal. However, recently a very different approach, which will be explained in this article, was adopted (section 6).

4 Basic Definitions

Before presenting the required semantics for concurrent contracts, used in MP-EIFFEL, a few basic definitions will be presented.

4.1 Concurrent Objects

A concurrent object is an object whose services might be requested by more than one processor in overlapping times, or in which the (direct) caller and callee processors might be different.

In SCOOP, concurrent objects are those attached to separate entities. In MP-EIFFEL, concurrent objects are those attached to shared or remote entities².

4.2 Concurrent Assertions

An assertion is said to be concurrent if it contains at least one concurrent assertion clause³. A concurrent assertion clause is one containing a concurrent boolean condition. Finally, a concurrent boolean condition is a boolean condition with a value that, in the context in which it is to be evaluated, may depend on the behavior of at least one processor other than the one testing it.

Assertions can be decomposed into two sets: a set of sequential assertion clauses, and another of concurrent assertion clauses.

As mentioned, in original SCOOP proposal only preconditions are expected to ever be concurrent assertions, and are always within routines with separate formal arguments. In MP-EIFFEL concurrent assertions are a little harder to track. The problem is that, as will be presented later, current MP-EIFFEL's policy for reserving objects is not bound to concurrent formal arguments. So, in

² A trigger call requires the use of a remote entity.

³ An assertion clause is a simple boolean condition declared inside an assertion.

MP-EIFFEL the presented general definition for concurrent assertions is the one required. Interestingly, in MP-EIFFEL the same assertion may behave sequentially or concurrently, depending on the context in which the client requests the routine. This property may allow instances of a class to be safely used either as sequential or concurrent objects.

5 The Behavior of Concurrent Assertions

Regardless of the concurrent language, if concurrent assertions are allowed to exist, how should they behave?

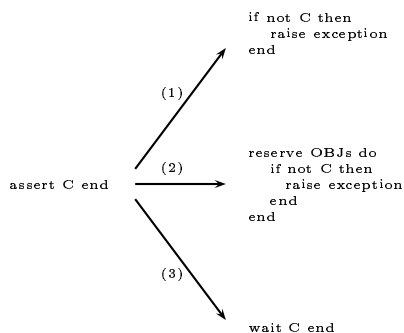


Fig. 1. Possible behaviors of concurrent assertions

Since, by definition, a concurrent assertion depends, at least, on a processor other than the one which is testing the assertion, its normal unsynchronized sequential behavior ((1) in figure 1) would clearly be a race condition, hence an unacceptable one.

Another possibility would be to unconditionally grab all the concurrent objects involved in the assertion (2), and then use the assertion as if it was sequential. This behavior is also a source of race conditions because once the object is reserved for the exclusive use of the processor responsible for testing the assertion, the concurrent assertion might be false depending on unpredictable timing relations between processors.

So, the only safe behavior is to attach concurrent assertions to wait conditions (3): a concurrent assertion causes its executing processor to wait until the concurrent objects are available and the concurrent assertion is verified. Not surprisingly, exactly the same behavior that is proposed for concurrent preconditions in SCOOP.

A very important aspect in which concurrent assertions differ from sequential ones, is the fact that their presence (as a wait condition) is essential to ensure program correctness. So, while correct sequential assertions can be safely disabled at runtime, concurrent assertions must be kept as conditional synchronization points.

6 Reserving Concurrent Objects

Equipped with a proper behavior for concurrent assertions we can now address the problem of concurrent object reservation.

In our opinion neither SCOOP's original approach, nor its most recent proposal, are the most appropriate ones. The requirement which limits qualified separate calls to be applied only to separate formal arguments, although with clear semantics, seems an overwhelming restriction. Also, the necessity to propagate concurrent preconditions to new routines with formal separate arguments increases the feeling that there is something not quite right. Finally, not only does this mechanism seem to be a redundant approach, but also, and most importantly, it does not prevent the erroneous situation of a missing concurrent precondition, just because the programmer forgot to duplicate it in the routine with separate formal arguments (following back to the (2) behavior of figure 1). As already mentioned, this type of concurrent errors are also race conditions, because there will be no logical relation between the action of object reservation and the desired assertion.

The problem is clearly a result of the artificial separation between the object reservation mechanism and the assertions which are fundamentally concurrent (in the concurrent object). The use of a redundant intermediate layer using a routine with concurrent formal arguments is only a workaround to the problem.

In our opinion the problem should be approached directly from within the meaning of the various programming constructs that might be negatively affected by concurrent objects, and not by a language construct –concurrent formal arguments– that is not, most of the times, directly connected to the problem (even when involving attached entities).

One of the candidate constructs was identified earlier in SCOOP: concurrent preconditions. A precondition is a condition expected to be observed when the routine initiates its execution, hence it would be safe behavior to reserve all the concurrent objects involved in those preconditions (regardless of being handled by concurrent formal arguments). The same reasoning also applies to concurrent invariants: all of the concurrent objects involved should be reserved throughout the routine's execution. Postconditions, on the other hand, should not possess this behavior. The reason why they shouldn't is the result of the expected causal behavior of programs. Postconditions express conditions that are expected to be verified after the execution of a routine, hence it does not make much sense to reserve concurrent objects to protect the use of objects after they have already completed their work (it would be a non-causal behavior).

Interestingly, concurrent assertions are not the only language construct whose semantics might be negatively affected by concurrent objects. A similar problem may occur in structured conditional and iterative instructions. Those instructions also ensure algorithm preconditions to the instructions they select (or iterate).

```
(1) if CONDITION then
(2)   ...
(3) else
```

```
(4) ...
(5) end
```

In this program sample one expects that at the beginning of (2) the value of *CONDITION* is true, and false at (4). Otherwise, the conditional instruction would be quite useless (worse than that, it would be unsafe).

If *CONDITION* is a concurrent boolean expression then the only way to ensure that expected behavior is to reserve the involved concurrent objects until the end of the conditional instruction.

A similar behavior is expected in iterative instructions:

```
(1) from
(2)  INIT
(3) until
(4)  CONDITION
(5) loop
(6)  ...
(7) end
```

At the beginning of (6), the value of *CONDITION* is expected to be always false, hence if there are any concurrent objects involved, they are required to be reserved for a proper safe behavior.

As a result of this object reservation policy, it is quite possible that the same assertion may be concurrent or not, depending on the context in which it is called. For example a precondition that is concurrent when the object is used directly, may become sequential if the object is used inside a conditional instruction that has already reserved the object. A possible practical implementation of this behavior is presented in the appendix A.

It is also important to note that this reservation behavior should not be confused with the required correctness behavior of wait conditions in concurrent assertions. Concurrent boolean expressions within conditional and iterative instructions are not wait conditions, because those instructions are part of the normal program behavior (so both boolean results, either true or false, are acceptable program values).

With these semantics we are convinced that the use of concurrent objects will not only behave as expected, but it will also prevent their useless reservation (hence increasing its concurrent availability and also probably reducing liveness problems such as deadlocks). The race conditions that may occur in SCOOP when a precondition is not replicated in the reservation routine are also prevented.

There is, however, a possible disadvantage of both this approach and the attached mechanism recently proposed for SCOOP, when compared with the original SCOOP proposal (in which all external object reservation was concentrated in the callee object interface). When external object reservation occurs in the concurrent object interface, the object is in a stable time. Hence, it remains usable by other processors while waiting for the availability of the other external objects (this situation was mentioned in section “Grabbing multiple objects” in [5]).

7 Final Remarks

This article briefly presents the ongoing work concerning the concurrent semantics of contracts, and the related problem of concurrent objects exclusive reservations.

Much work remains to be done, not only in what concerns extended practical testing of these mechanisms, but also to achieve a complete and coherent definition of the semantics of the language.

8 Acknowledgments

Many thanks to João Rodrigues and Tomás Oliveira e Silva for their invaluable help and suggestions.

References

1. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
2. B. Meyer. Design by contract. In *Advances in Object-Oriented Software Engineering*, pages 1–50. Prentice Hall, 1992.
3. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.
4. B. Meyer. Attached types and their application to three open problems of object-oriented programming. In *ECOOP 2005, Proceedings of European Conference on Object-Oriented Programming*, pages 1–32. Springer Verlag, 2005.
5. M. Oliveira e Silva. Concurrent object-oriented programming: The MP-Eiffel approach. *Journal of Object Technology: Special issue: TOOLS USA 2003*, 3(4):97–124, April 2004.
6. M. Oliveira e Silva. Automatic realizations of statically safe intra-object synchronization schemes in MP-Eiffel. (submitted for publication) Draft version available at <http://www.ieeta.pt/~mos/pubs>, 2006.

A Concurrent Assertions Detection

In MP-EIFFEL the same assertion may be concurrent or sequential, depending on the context of its utilization. The following algorithm solves this problem.

It is well known that in concurrent programs one cannot test if a concurrent object is being reserved by any other processor. However, it is perfectly safe to test if the current processor owns the exclusive right to use an object, if nothing is expected to be done –other than a conditional wait– if it doesn't.

An assertion which may not be sequential due to a concurrent object *obj*:

```
assert C(obj) end
```

can be safely implemented with the following algorithm:

```
(1) if mylock(obj) then -- sequential assertion
(2)   if not C then
(3)     raise exception
(4)   end
(5) else -- concurrent assertion
(6)   wait C end
(7) end
```

Of course this algorithm should only be applied to assertions that may be concurrent. In MP-Eiffel those assertions are required to use qualified calls to shared or remote entities.