Contracts for concurrency

Piotr Nienaltowski¹, Bertrand Meyer² and Jonathan S. Ostroff³

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:

\[
\begin{align*}
\{ INV \land Pre_r \} & \quad body_r \quad \{ INV \land Post_r \} \\
\{ Pre_r[\sigma/\bar{f}] \} & \quad x.r(\sigma) \quad \{ Post_r[\sigma/\bar{f}] \}
\end{align*}
\]

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 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.