Symposium on Concurrency, Real-Time, and Distribution in Eiffel-Like Languages

Proceedings

4-5 July 2006

York, United Kingdom

Organised by: the ARTIST Network of Excellence, the University of York, and the University of Teesside

Editors: Richard F. Paige and Phillip J. Brooke

Preface

This volume contains the papers presented at the First International Symposium on Concurrency, Real-Time, and Distribution in Eiffel-like Languages (CORDIE 2006), held 4-5 July, 2006, at the King's Manor, York, United Kingdom.

The symposium's focus was on evolving concurrency, real-time and distribution mechanisms in object-oriented languages that supported features like those appearing in Eiffel. In particular, we saw a substantial emphasis on the relationships between Eiffel-like contracts (i.e., routine pre- and postconditions, and class invariants), and mechanisms for concurrency such as synchronisation and locking. Topics of interest included contracts for concurrency, real-time extensions, formal models and semantics for concurrency in Eiffel-like languages, tool support, distributed middleware, synchronisation, locking, asynchronous exception handling, and formal verification.

The programme committee accepted 9 full papers for presentation, some after a second round of review and revision. Each submission was reviewed by at least three members of the international programme committee. We are grateful to the programme committee members for their timely completion of the reviewing process, and for the quality and detail of their reviews and discussion.

Our thanks go to all members of the programme committee for their efforts; the authors, for submitting their papers; our invited speakers, Andy Wellings and Antónia Lopes; our closing speaker, Bertrand Meyer; our sponsors: the ARTIST Network of Excellence, the University of Teesside, the University of York, and the Chair of Software Engineering, ETH Zürich. We also thank Formal Methods Europe for their support in advertising the event.

June 2006

Richard F. Paige
Phillip J. Brooke (Program Co-Chairs)

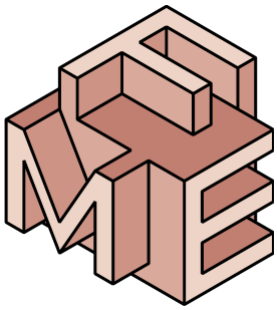
Organisation

CORDIE'06 is organised by the University of York and the University of Teesside.

Programme Committee

Simon Dobson	UC Dublin, Ireland
Jin Song Dong	National University of Singapore
Chris Gill	Washington University, USA
Michael Gonzalez Harbour	University of Cantabria, Spain
Jeremy Jacob	University of York, UK
Jeff Magee	Imperial College, UK
Bertrand Meyer	ETH Zürich, Switzerland
Piotr Nienaltowski	ETH Zürich, Switzerland
Jonathan Ostroff	York University, Canada
Jan Vitek	Purdue University, USA
Alan Wood	University of York, UK
Jim Woodcock	University of York, UK

Sponsors and Supporters



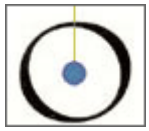


Table of Contents

RECOOP: Real-Time Concurrent Programming with Eiffel

Andy Wellings
Department of Computer Science, University of York, UK.
andy@cs.york.ac.uk

Abstract. Although Eiffel has been around since the 1980s, and there have been various attempts to introduce concurrency into the language, none of these attempts have been successfully extended to support real-time system development. This is due, in part, to the concurrency models not having an easy way to identify threads, which in turn complicates the expression of real-time attributes. The requirement to adhere to Eiffel's strict contract model also increases the difficulty of providing mechanisms in support of major real-time requirements (such as asynchronous notification). Using the SCOOP concurrent version of Eiffel as a starting point, this presentation illustrates the problems of adding real-time support. We show that changing the notion of "separateness" allows a more traditional real-time model to be extracted from the program (without having to have different concepts for objects and threads). The result is an integrated concurrent object-oriented language, called RECOOP, that also allows the convenient association of real-time attributes.

In moving from a sequential OOP language to a concurrent OOP language it is usual to reinterpret preconditions as "wait conditions". This talk argues that in moving from a concurrent OOP language to a real-time concurrent OOP language it is also necessary to reinterpret postconditions. RECOOP allows the failure of postconditions to be reported as early as possible. This mechanism when used with time primitives allow deadlines misses to be detected immediately. If a precondition can be reinterpreted as "delay the start", a postcondition can be reinterpreted as "hasten the completion".

An important requirement for real-time systems is to be able to change dynamically real-time attributes in a responsive manner. The talk illustrates that this requirement cannot be met without more fundamental changes to the Eiffel model. Building on the recent introduction of "frames" into the ECMA Eiffel standard, we propose an "allow" clause, which allows the concurrent execution of an object's methods as long as pre and postconditions are not violated.

System Design in CommUnity
— A Categorical Approach —

Antónia Lopes

Department of Informatics, Faculty of Sciences, University of Lisbon
Campo Grande, 1749–016 Lisboa, Portugal,
mal@di.fc.ul.pt

CommUnity was born, more than 10 years ago, as a language for parallel program design similar to Unity and Interacting Processes. The goal, at that time, was to show how programs fit into Goguen's categorical approach to General Systems Theory. Since then, the language and the design framework have been extended in order to provide a formal platform for the architectural design of open, reactive, distributed and reconfigurable systems.

The locality of names that is intrinsic to Category Theory forces components to be designed in CommUnity without explicit references to other components and, hence, enforces the principle that any interconnection between components must be explicitly established. In this way, the distinctive feature of the language became the emphasis that it puts in the externalisation and explicit modelling of interactions as first-class citizens. The language supports the complete separation of coordination from computation concerns, providing a paradigmatic architectural description language in which connectors are first-class entities. Systems are described through configurations built from components and connectors and evolution can be achieved through the addition, deletion or substitution of the connectors that coordinate the interaction between their components, without interfering with the computations that are performed locally.

More recently, CommUnity was extended in order to support the description of the distribution and mobility dimension of systems. In the developed extension, the process of integrating and managing mobility in architectural models of distributed systems is not intrusive on the options that are made at the level of the other two dimensions, meaning that a true separation of concerns between computation, coordination and distribution can be enforced at the level of architectural models.

This talk will review the main features of CommUnity language and discuss how its categorical semantics, being largely language-independent, can be applied to other languages.

SCOOP: a Retrospective and Prospective

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

Abstract

In this talk I will present a retrospective on SCOOP, and will also look forward to future efforts, some of which focus on issues and complications that have been presented at CORDIE'06.

Verifying Properties beyond Contracts of SCOOP Programs

Jonathan Ostroff, Faraz Ahmadi Torshizi, and Hai Feng Huang

Department of Computer Science and Engineering, York University,
4700 Keele St.,Toronto, ON M3J 1P3, Canada
{jonathan, faraz, hhuang}@cs.yorku.ca

Abstract. SCOOP and Spec# are programming languages that aim to extend Design by Contract to concurrent and reactive systems. In this paper we discuss how appropriate theorem provers (using Hoare-like verification) can be used to statically check that the contracts are obeyed in concurrent executions, as well as discussing the syntactic and semantic differences between SCOOP and Spec#. We provide a formal model for SCOOP programs as a fair transition system and we use temporal logic for describing system properties beyond contractual correctness. We show that verified contracts provide only a certain measure of correctness, but may not be able to guarantee additional safety and liveness system properties without global reasoning. We show how Microsoft Research's SpecExplorer tool can be used to test SCOOP programs for system properties beyond contracts.

1 Introduction

Concurrent and reactive systems are hard to write and even harder to test. In industrial settings, software verification consists almost entirely of testing. Testing is one of the costliest and most laborious aspects of commercial software development, especially given the lack of systematic engineering methodology, clear semantics and adequate tool support. Concurrency and the need to develop software for reactive systems introduces a level of complexity beyond that of sequential programming. Object-oriented code with dynamic thread creation also introduces additional levels of complexity.

Formal methods using model-checkers and theorem provers have not been considered practical for software applications, but this situation is slowly changing. Until relatively recently, the majority of the work carried out by the formal methods community for proving programs correct has been devoted to special languages that differ from industrial strength programming languages [21]. This is a useful phase as it allows the formal methods community to experiment with new methods.

Recently, more steps have been taken to work with real programs written in modern programming languages. The B-method was used to produce the control system for the Paris driverless metro [3]. In this system, the specification was written and refined from the B specification language into Ada code with all

the refinements checked via a theorem prover. Abstract interpretation has been used in [8] to analyze some C programs of up to 100K lines of code, although there are difficulties dealing with rich data structures and dynamic threads. Java PathFinder (JPF) is a verification environment for Java for detecting deadlocks and assertion violations integrating program analysis and model checking [9]. The following quote from [21] is instructive:

> Although it is hard to quantify the exact size of program that JPF can currently handle - "small" programs might have "large" state-spaces - we are routinely analyzing programs in the 1000 to 5000 line range. ... it is naive to believe that model checking will be capable of analyzing programs of 100000 lines or more ...

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

The authors of [4] investigate the use of contracts in object oriented code. The authors state that contracts are known to be a useful technique to specify the precondition and postcondition of operations and class invariants, thus making the definition of object-oriented analysis or design elements more precise. The paper shows how to reuse and instrument contracts to ease testing. A thorough case study is run where they define contracts, instrument them using a commercial tool, and assess the benefits and limitations of doing so to support the isolation of faults. They show that Design by Contract (DbC) has proven to be a powerful lightweight method for documenting contracts in object oriented code as well as for detecting bugs.

The object oriented Eiffel programming language is an industrial strength language with a mature contracting mechanism [13]. ESC/Java [19] shows how to add and check contracts for Java and Spec# is a superset of C# which has a contracting mechanism as well as static verification of contracts [2].

The Simple Concurrent Object-Oriented Programming (SCOOP; hereafter "Scoop") mechanism was proposed as a way to introduce inter-object concurrency into the Eiffel programming language [13]. The mechanism extends the Eiffel language by adding one keyword **separate** that can be applied to entities (attributes and formal routine arguments). If entity `e` is declared **separate** then any call `e.f` is executed in its own thread of control; application of **separate** to entities or arguments indicate that these constructs are points of synchronization.

Part of the Scoop mechanism was implemented by Compton [6] by building upon the GNU SmartEiffel compiler and runtime system, and a Scoop translator using the Eiffel Software compiler was reported in [7]. Scoopli is currently the most up-to-date implementation of Scoop. Using a library approach and the Eiffel Software compiler, code runs as a C or .NET executable [15].

In this paper we will describe the Scoop mechanism via a simple example called *Zero-One* and compare Scoop and Spec# especially with respect to static

verification of contracts using theorem provers. We will contrast runtime Assertion Checking versus static Verification, and we will show that contracts can be used to detect certain classes of errors. However, we will also show that there are system properties that contracts alone (without global reasoning) may not detect. We provide an outline of how to convert Scoop code to fair transition systems and we use temporal logic for writing system specifications. We show how to build reduced models and how to use SpecExplorer for testing system properties beyond contracts. The combination of contracts and reduced model testing provide lightweight formal verification that scales up to large systems.

2 Sequential and Concurrent Computation

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

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

A *subsystem* is a processor together with the set of objects it performs actions on. Within a subsystem, communication is synchronous, and execution follows the usual Eiffel sequential model. Communication between subsystems is asynchronous and processing is in parallel. This potential parallelism is the result of different processors handling each subsystem [6].

A separate object is any object that from the viewpoint of the current object is in a different subsystem. At run time, a separate object can only be referenced (if reachable at all) through a separate entity. An entity is either an attribute of a class, a formal argument of a routine, or a local variable of a routine. A separate reference is a reference to a separate object. This reference must be through a separate entity that is not void, and not attached to a local object. A separate call is any routine call $t.r(x)$, from the current object in which the call is made, where the target t is a separate object. A subsystem is created with the creation of a separate object.

2.1 A Simple Sequential Example

To motivate the main discussion we describe a simple Scoop program – the Zero-One example which uses a sequential class DATA (Fig. 1) written in standard Eiffel. The contracts (preconditions, postconditions and class invariants) document the specification and may also be used to find implementation bugs and demonstrate the correctness of the code.

Correctness of the implementation can be demonstrated either by run-time *Assertion Testing* or by static compile-time *Formal Verification* via the use of a theorem prover. Consider the code in class TEST (Fig. 2) which uses class DATA.

The `create d` instruction (in routine `r`) does a default initialization of all the attributes as shown in the immediately following check statement. Will the feature call `d.one` in the above code succeed without contract violations? The correctness rule for a general feature call $t.r(x)$ is:

$$\frac{\{pre_r \wedge I\} do_r \{post_r \wedge I\}}{\{pre'_r\} t.r(x) \{post'_r\}} \quad [\text{ CR1 – Sequential Correctness Rule }]$$

where pre_r, do_r and $post_r$ are the precondition, body and postcondition of routine r respectively and I is the invariant of the class in which r occurs. The primed notation used in the consequence of rule [CR1] refers to the contracts suitably qualified to the target t. For example, for routine `one` of class DATA, rule CR1 reduces to

$$\frac{\{x = 0 \wedge y = 0\} x := 1;\ y := 1;\ c1 := c1 + 1 \{x = 1 \wedge y = 1 \wedge Q\}}{\{d.x = 0 \wedge d.y = 0\} d.one \{d.x = 1 \wedge d.y = 1 \wedge Q'\}}$$

where $Q \stackrel{\text{def}}{=} c0 = \textbf{old}\ c1 \wedge c1 = \textbf{old}\ c1 + 1 \wedge b = \textbf{old}\ b$ and $Q' \stackrel{\text{def}}{=} d.c0 = \textbf{old}\ d.c0 \wedge d.c1 = \textbf{old}\ d.c1 + 1 \wedge d.b = \textbf{old}\ d.b$.

In Formal Verification, we can use a theorem prover to check each routine for the verification conditions generated by rule [CR1]. Such a static check guarantees that the code will run correctly without contract violations at runtime. We have implemented such a theorem prover for a significant subset of sequential Eiffel [18, 20]. This theorem prover trivially verifies the correctness of DATA (Fig. 1) and the correctness of the routine `r` in class TEST (Fig. 2). The theorem prover is putatively sound (on the assumption that it is constructed correctly) but not complete. The theorem prover will issue a warning if a verification condition fails to prove with some debugging information as to the source of the problem. The warning could indicate a real bug, but could also mean that the verification condition is true, but that the theorem prover was unable to prove it. Manual intervention would then be required to achieve full certification.

In Assertion Testing, we enable run-time assertion checking and the compiler then generates code that checks the contracts at each feature call (such as `d.one`). Assertion Testing is much weaker than Verification, as [CR1] is only checked for the executions in our testing suite. However, any code of any size can be automatically checked in this manner without the need to provide complete contracts. Testing is thus a successful totally automated lightweight method for documenting and automatically checking specifications.

2.2 A Simple SCOOP Example Using DATA

Classes `ZERO` and `ONE` show some of the main Scoop properties (Fig. 3). For the purposes of this discussion, we assume that a single instance of `ONE` is running

```
class DATA feature
  x,y,c0,c1: INTEGER
  b: BOOLEAN

  zero is
    require x = 1 and y = 1
    do
       x:=0; y:=0; c0 := c0 + 1
    ensure
      x = 0 and y = 0
      c0 = old c0 + 1 and b = old b and c1 = old c1
    end

  one is
    require
      r1: x = 0 and y = 0
    do
       x:=1; y:=1; c1 := c1 + 1
    ensure
      e1: x = 1 and y = 1
      e2: c1 = old c1 + 1 and c0 = old c0 and b = old b
    end

  stop is
    do
      b := true; x := 2
    ensure
      b and x = 2
      y = old y and c0 = old c0 and c1 = old c1
    end
invariant
    inv_data: ((x = 0 and y = 0) or (y = 1 and x = 1)) or b
end -- class DATA
```

Fig. 1. Class DATA

```
class TEST feature
  d: DATA

  r is
    do
      create d
      check
        d.x = 0 and d.y = 0 and d.b = false and d.c1 = 0 and d.c0 = 0
      end
      d.one
    end
end
```

Fig. 2. Class TEST

in a subsystem under the control of processor π_1. Likewise an instance of `ZERO` is running under the control of processor π_0.

Class `ROOT` is shown in the listing in Fig. 4. A system execution is initiated when the constructor `ROOT.make` is called. The constructor creates and initiates the execution of the three subsystems π_0, π_1 and π_d.

In class `ONE`, attribute `data` of type `DATA`s is declared **separate**. This means that the object attached to `data` at runtime runs in its own subsystem (e.g. under the control of processor π_d) and thus under a different processor than the one handling the current object. The responsibility of routine `run` is to invoke the separate call `data.one` repeatedly. Scoop requires that such calls be wrapped in a routine such as `do_one` (see lines 16 and 20).

It is instructive to follow an execution that has arrived at line 16 (which we denote as $\pi_1 = 16$). Control transfers to line 20 where processor π_1 waits to get a lock on the data object under control of π_d. If deadlock does not occur and the lock is obtained (with unique access to `data.b`), the non-separate precondition $count \leq 1000$ is immediately checked at line 23. A failure generates a precondition exception, and success means that π_1 *waits* for the separate precondition $\neg\ data.b$ at line 22 to become true. It is thus possible for this subsystem to deadlock at line 22 if the condition never becomes true. Assuming the wait condition $\neg\ data.b$ becomes true, execution continues at line 27. An asynchronous feature call `data.one` is sent to subsystem π_d, and execution continues until 29 where π_1 waits for all asynchronous calls to terminate including the query `data.x = 1`, at which point the assignment can be executed (this is called wait by necessity [13]).

There is another danger. The asynchronous separate feature call `data.one` at line 27 may fail when it is finally executed by π_d because the non-separate precondition of `DATA.one` (i.e. $data.x = 1 \wedge data.y = 1$) may fail to hold (this condition was not checked by the π_1 client prior to the feature call). There are thus a variety of reasons why this Scoop program may fail:

1. Deadlocks may occur at lines 20 [call this failure F1] and 22 [F2].
2. The non-separate precondition may fail at line 23 (a client check is not performed at line 16) [F3].
3. The non-separate precondition of `DATA.one` may fail at line 27 (or more correctly, the failure will occur when the precondition is checked in subsystem π_d) [F4].

Although we described the execution in terms of acquiring and releasing locks, it is the job of the Scoop compiler to enforce the atomicity described above. The compiler will automatically detect where the Scoop **separate** keyword is missing or inappropriately used, and properly enforce the appropriate behaviour. Thus many race conditions are automatically eliminated. This does not mean that all race conditions are eliminated. The claim that, by using the Scoop model, we eliminate many bugs that come from race conditions, is like the claim that functional languages eliminate side-effect bugs. We may still write code such that the same kind of interference occurs in both cases, but the language leads you naturally away from it.

```
class ONE create
01   make
02 feature
03   data: separate DATA
04   count: INTEGER
05
06   make(d: separate DATA) is
07     do
08       data := d
09     end
10
11   run is
12     do
13       from
14        until false -- later changed to count > 1000
15       loop
16         do_one(data)
17       end
18     end
19
20   do_one(d: separate DATA) is
21     require
22       separate_pre: not d.b
23       non_separate_pre: count <= 1000
24     local
25       test: BOOLEAN
26     do
27      d.one
28      count := count + 1
29      test := d.x = 1
30     ensure
31       non_separate_post: count = old count + 1
32       separate_post: d.x = 1 and d.y = 1 and d.b = old d.b
33     end
end -- class ONE
```

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

```
class ROOT create
    make
feature
  d:  separate DATA
  p0: separate ZERO
  p1: separate ONE

  make is
    do
      create d
      create p0.make(d)
      create p1.make(d)
      run(p0, p1)
    end

  run(z: separate ZERO; o: separate ONE) is
    do
      z.run
      o.run
    end
end
```

Fig. 4. Class ROOT – initiates the three subsystems

3 Detecting Contract Failures

How can we detect deadlocks [F1, F2] and contract failures [F3, F4] as described in the previous section? Due to interference from other subsystems, our formal condition for class correctness must now change to the following [13] (page 1023):

$$\frac{\{pre_S \wedge pre_{NS} \wedge I\}\, do_r \{post_S \wedge post_{NS} \wedge I\}}{\{pre'_{NS}\}\, t.r(x) \{post'_{NS}\}} \quad [\text{ CR2 Rule }]$$

where pre_S and $post_S$ are separate pre/postconditions and pre_{NS} and $post_{NS}$ are non-separate pre/post conditions. There is a significant difference between the sequential rule [CR1] and the [CR2] rule. The sequential rule is a full correctness condition – if the antecedent holds, then not only is the call partially correct, but it is also guaranteed to terminate. By contrast, the [CR2] rule only checks partial correctness as it does not incorporate any checks that would catch deadlocks such as [F1] and [F2]. To detect such deadlocks, we need information about other subsystems. We will examine safety properties such as system deadlock detection and liveness properties in a later section.

We may use the [CR2] rule to detect contractual errors such as [F3] and [F4]. Consider first [F3]. If we change our theorem prover to use [CR2] rule instead of [CR1], then we obtain a warning at line 16 (Fig. 3) because we are calling `do_one` without satisfying its precondition $count \leq 1000$ at line 23. Likewise for

[F4], we obtain a warning at line 27 because we are calling `DATA.one` without guaranteeing its precondition.

We can eliminate these warnings from the theorem prover by strengthening the code. [F3] can be fixed by changing the loop guard at line 14 to `(until count > 1000)`. [F4] can be fixed by using a stronger separate precondition at line 22: `not d.b and d.x = 0 and d.y = 0`. This strengthened precondition means that π_1 waits at line 22 until some other processor (e.g π_0) sets the data variables to zero. With these changes, a theorem prover using [CR2] rule will pass without warnings.

We tested the original and revised code in Scoopli. The original code failed with contract exceptions [F3] and [F4], and the revised code passed, thus illustrating Assertion Testing. However, neither Formal Verification nor Assertion Testing were able to guarantee detection of deadlocks such as [F1] and [F2].

4 Comparison of Spec# and Scoop

The Spec# programming system[1] extends C# with contracts (like Eiffel), while it also aims to support concurrency based on object ownership [11]. Spec# extends the type system of C# to include non-null types and checked exceptions. It provides method contracts in the form of pre/postconditions as well as object invariants. The Spec# compiler is integrated into the Microsoft Visual Studio development environment for the .NET platform. The compiler statically enforces non-null types, and emits run-time checks for method contracts and invariants [2].

The Spec# static program verifier (Boogie) generates logical verification conditions from a Spec# program. Internally, it uses an automatic theorem prover (currently *Simplify*) that analyzes the verification conditions to prove the correctness of the program or to find errors in it. Spec# aims to maintain invariants in object-oriented programs in the presence of callbacks, threads, and inter-object relationships [11].

According to [1] there is a problem with the normal rule for invariants especially for concurrent programs. The authors of [1] write that a popular view is that an object invariant is simply a shorthand for a postcondition on every constructor and a pre/postcondition on every public method. The idea behind this view is that an object's invariant should hold whenever the object is publicly visible. This view in itself is appropriate, but is often combined with the following faulty regime. Callers of the methods of a class T do not need to be concerned with establishing the implicit precondition associated with the invariant. For the invariant of a class T to hold at entries to its public methods, it is sufficient to restrict modifications of the invariant to methods of T and for each method in T to establish the invariant as a postcondition. This regime permits a method to violate an object invariant for the duration of the call, as long as it is re-established before returning to the caller. But, unless every method body is

[1] http://research.microsoft.com/specsharp

```
public sealed class One {
   [LockProtected]
   public Data ! data;
   public int count=0;

   invariant data != null;

   public One([LockProtected] Data d)
      requires d != null;
   {
      data = d;
   }

   public void Run()
       ensures (data.x == 0 && data.y == 0 && !data.b && count <= 1000)
           ==> (count == old(count) + 1);
       ensures (data.x == 0 && data.y == 0 && !data.b && count <= 1000)
           ==> (data.x == 1 && data.y == 1 && data.b == old(data.b));
   {
       while (count <= 1000)
          invariant this.IsExposable;
       {
         expose (this)
         {
             assume data.IsLockProtected;
             acquire (data)
             {
                 if (data.x == 0 && data.y == 0 && !data.b)
                 {
                     assume data.IsPeerConsistent;
                     //Console.WriteLine("1 count: " + count);
                     count++;
                     data.One();
                     assert data.x == 1;
                 }
             }
          }
       }
   }
}
```

Fig. 5. Spec# Code similar to ONE

atomic, this is a problem as illustrated in the paper for a routine that calls itself at a point where the invariant has not yet been re-established. For Spec# the recommendation is made for a construct that declares that the invariant may be temporarily violated (see for example the `expose` construct in Fig. 5).

The problem identified by [1] needs to be examined in the Scoop model. A call to a separate routine is atomic, thus ensuring that no other subsystem will interfere. Second, the Scoop (and Eiffel) model only require the invariant to hold on entry to a *qualified* call (see [13], page 366). There is no such rule for unqualified calls which are not directly executed by clients but only serve as auxiliary tools for carrying out the needs of qualified calls. In such cases it is in order to temporarily violate the invariant provided it is re-established at the end of the routine. We refer the reader to [14] for further discussion.

The Spec# static program verifier uses a theorem prover to prove rules such as [CR1]. The verifier is interesting but still very much in the experimental stage. For example, verification of genericity and inheritance are not yet fully implemented, some primitive types such as reals are not checked, and pure methods in postconditions do not verify. Spec# code (approximately) equivalent to the revised version of class `ONE` (Fig. 3) is shown in Fig. 5. There is not yet much documentation available to enable us to fully evaluate the tool. As far as we were able to determine, there are a number of differences when compared to Scoop.

1. Atomicity is enforced by explicit acquire statements, and various constructs such as sealed assertions and lock protection must be declared.
2. In Spec# all calls are synchronous. Scoop offers a mix of synchrony and asynchrony.
3. To get the theorem prover to work, various assumptions must be added (see `assume` clauses).
4. Preconditions are correctness conditions, not *wait* conditions as in Scoop. This means that wait conditions must be explicitly programmed in via wait constructs.
5. Basic Spec# (without the extensions of [11]) allows object sharing between multiple threads, resulting in potential intra-object concurrency and races that Scoop would prohibit.

Impressively, the Spec# verifier was able to prove the correctness of the contracts and catch incorrect implementations such as those associated with [F3] and [F4]. Also, the ability to statically check for non-null types is significant. However, we still lack full automated capabilities to detect system properties such as complete deadlock detection and liveness properties.

5 SCOOP semantics for contracts

As mentioned earlier, the [CR2] rule, while being correct, is too weak to allow us to argue about separate postconditions. Further, while the Scoop model treats separate preconditions as wait conditions (rather than correctness conditions), we have not explained how invariants and postconditions are treated. Recently,

Piotr Nienaltowski and Bertrand Meyer have proposed that postconditions be treated as wait conditions (similar to that of preconditions) and that invariants be disallowed from referring to separate entities [17]. In the sequel we follow the lead of [17] with respect to the intuitions behind postcondition and invariant semantics but provide a temporal logic description of the semantics. Rule 1.5 in [17] is not strong enough to allow for fully compositional proofs of correctness and liveness. This paper will provide a temporal logic version of the semantics based on recent discussions with the authors of [17] and to be reported more fully in [16].

It is convenient to use temporal logic to describe system properties (beyond contracts between a client object and a supplier object). So as to describe the contracting semantics and system properties we provide a schema of how to translate Scoop programs into fair transition systems, which can then be used as the basis for expressing temporal logic system properties. We provide below the main features of a fair transition system in the sense of Manna and Pnueli [12], and we provide a sketch of how to adapt fair transition systems for Scoop programs.

Fair Transition Systems

A fair transition system M is a 5-tuple $M = (V, I, T, J, F)$;

1. The *system variables* V is a finite set of typed variables. The creation of a new subsystem (e.g. `create p1.make(d)` in Fig. 4) corresponds to extending V with a corresponding control variable (e.g. π_1 which is the handler for this instance of class `ONE`). A control variable for a subsystem can be used to indicate which line of code in that subsystem is currently being executed (it is never used in actual program text). A state s of the system is a mapping that assigns to each variable $v \in V$ a value in $type(v)$. The set of all states is denoted by Σ.
2. The *initial condition* I is a boolean valued expression in the variables that characterizes the states at which the execution of the system can begin. A state s satisfying I, i.e. $s \models I$, is called an *initial state*.
3. T is a finite set of transitions. Each transition τ in T is a function $\tau : \Sigma \rightarrow 2^{\Sigma}$ that maps a prestate s in Σ to a (possibly empty) set of τ-*successor* poststates $\tau(s)$ which are obtained when τ is taken. Each state s' in the set $\tau(s)$ is defined to be a τ-*successor* of s. The transition relation $\rho(\mathbf{old}\, V, V)$ describes a set of 2-tuples (consisting of a prestate s and poststate s') that relates the prestate s to its τ-successor $s' \in \tau(s)$, and where $\mathbf{old}\, V$ (by which we mean any of the variables in V) is evaluated in the prestate s, and V is evaluated in the successor state s'.
4. $J \subseteq T$ is a set of *just* transitions. If a just transition $\tau \in J$ is continually enabled it must eventually be taken. Likewise F is a set of fair transitions, i.e. a fair transition that is enabled infinitely often must eventually be taken.

Executions of Fair Transition Systems

An *execution* of a model $M = (V, I, T, J, F)$ is any infinite sequence of states:

$$\sigma = s_0 \xrightarrow{\tau_0} s_1 \xrightarrow{\tau_1} s_2 \xrightarrow{\tau_2} \dots$$

with $\tau_0, \tau_1, \tau_2 \dots$ elements of T, so that the following three requirements are satisfied:

1. **Initialization:** The first state of the execution satisfies the initial condition, i.e. $s_0 \models I$.
2. **Succession:** For all positions i in the execution, $s_{i+1} \in \tau_i(s_i)$, i.e. state s_{i+1} is a τ_i-successor of state s_i using the transition relation. This also means that $s_i \models e_{\tau_i}$ where e_{τ_i} is the enabling condition (conjunction of separate and non-separate preconditions and locations) of transition τ_i. We say that τ_i is *taken* at position i in the execution σ, and we may write $taken(\tau_i)$ to express this.
3. **Justice and Fairness:** For each τ in the justice set, it is not the case that τ is continually enabled beyond some position in the trajectory, but taken at only finitely many positions in the execution. A similar constraint applies for fair transitions.

Scoop code can be converted to a fair transition system using the techniques in [12]. Each construct such as assignments and alternatives are translated into transitions, and the transitions of different subsystems are interleaved with each other, with the fairness constraints removing the non-fair executions. The considerations below may be used to translate the feature call into appropriate transitions.

Postconditions and Invariants

As mentioned earlier, the Scoop model treats preconditions as wait conditions. We now need to consider invariants and postconditions.

There is a major difference between the feature call `Current.do_one(data)` at line 16 and feature call `d.one` at line 27 in Fig. 3.

- At line 16, the argument `data` is in an *unlocked* context because the enclosing routine `run` does not declare `data` as separate.
- At line 27, the feature call `d.one` is in a *locked* context because `d` is locked by the enclosing routine `do_one`. Even if routine `one` would have an argument, e.g. `d.one(x)`, it would be considered as executing in a locked context if both `x` and `d` are declared separate in the formal argument list of `do_one`. The locked case can use the standard sequential [CR1] rule.

Consider, now, the unlocked case at line 16 in Fig. 3, at which point the handler π_1 must execute the routine `do_one(data)` where `data` is an attribute declared as `separate DATA`. This routine "wraps" all accesses to `data` within

one call so that no other processor may interfere. Handler π_1 waits to acquire a lock on `data` and for the precondition of the routine `do_one` to become true. Who manages lock acquires and releases and who is responsible for executing the body of the routine `do_one`?

We may assume that the Scoop runtime has a global handler π that manages an action queue for servicing separate calls such as `do_one`. Separate feature calls are queued in the order received, and the global handler guarantees that calls are handled in that order. Provided that all calls can be shown to terminate[2], the global handler guarantees that *Acquire*(`data`) eventually becomes true at line 16. The global handler is solely responsible for managing all locks on sperate subsystems, granting them and releasing them as required. If, also, the precondition *Pre*(`do_one`) subsequently becomes true, then π can mark `do_one` as currently executing and then initiate execution of the body of `do_one`.

In the mean time, handler π_1, having (asynchronously) handed responsibility for the routine call `do_one` off to global handler π, may continue executing at line 17. In the actual example, it just loops back, but in general it could do some local processing before returning to line 16.

Now, routine `do_one` has a postcondition (see lines 31 and 32). When is the postcondition evaluated and by whom? It cannot be evaluated by π_1 immediately after dispatching `do_one` to π because then there would be unnecessary blocking at line 16, which would undermine our attempts at being able to process locally while `do_one` is executed elsewhere.

The logical candidate to choose is handler π (who is anyway responsible for handling `do_one` and lock acquisitions and releases). Handler π checks that `do_one` has completed processing, checks the postcondition, flags any contractual exceptions, and releases the lock on `data`. Consider handler π_1 executing the wrapped routine `do_one` as follows:

```
15:
16: do_one(data)
17:
```

We use our temporal logic framework to describe the behaviour of routine `do_one`.

$$\begin{aligned} \Box[\ (\pi_1 = 16) \wedge P \ &\rightarrow\ (\pi_1 = 16)\,\mathcal{U}\,(\pi_1 = 17)\ \wedge\ \Diamond(Q)\ ] \qquad (1)\\ P \ &\stackrel{\text{def}}{=}\ \Diamond[\mathit{Acquire}(\texttt{data}) \wedge \mathit{Pre}(\texttt{do\_one}) \wedge \mathit{Inv}]\\ Q \ &\stackrel{\text{def}}{=}\ \mathit{Acquire}(\texttt{data})\,\mathcal{U}\,[\mathit{Post}(\texttt{do\_one}) \wedge \mathit{Release}(\texttt{data}) \wedge \mathit{Inv}] \end{aligned}$$

where *Acquire*(*data*) and *Release*(*data*) are functions of the global handler π. *Pre* and *Post* stand for precondition and postcondition respectively of `do_one` and *Inv* is the invariant of class `ONE`. $\Diamond$ is the standard *eventually* operator, $\Box$ the

[2] e.g. using the sequential rule [CR1]. This would require one of the lock passing mechanisms to be implemented so that callbacks do not cause deadlocks.

henceforth operator, and $p\,\mathcal{U}\,q$ the *until* operator ("p *until* q" means eventually q, and p holds continuously at least until the first occurrence of q).

As in [17], it is not necessary for a separate postcondition to hold immediately after the execution of the routine's body. The wait semantics applies to postconditions but waiting happens on the supplier side, or more precisely it is managed by the global handler π and the data supplier. The separate target `data` is not released until the postcondition is satisfied.[3]

Note that (1) reduces to a much simpler form for the feature call `d.one` at line 27 because `one` (and any arguments of `one` should there be any) are all already in a locked context. Thus we may drop the lock acquisitions and releases and for such a routine call we may use the standard sequential rule [CR1].

What about invariants? As stated in [17], invariants play an important role in the Design by Contract methodology. They are the primary tool for ensuring the consistency of objects. To prove the correctness of a routine, we assume the invariant before the execution of the body and we must guarantee that it holds again when the body terminates. Rule (1) follows this pattern. 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, we prohibit the use of 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 reduce the wait semantics to a 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.

Temporal Logic System Specifications

With a fair transition model of Zero-One in place we may document system liveness and safety properties using temporal logic, e.g.

Specification S1: $\Box\Diamond(zero \wedge \bigcirc one)$ where $zero \stackrel{\text{def}}{=} (\pi_d.x = 0 \wedge \pi_d.y = 0)$ and $one \stackrel{\text{def}}{=} (\pi_d.x = 1 \wedge \pi_d.y = 1)$. [S1] asserts that we alternate between *zero* and *one* infinitely often. This liveness property is false because the loop only executes 1000 times.

Specification S2: $\Box(\pi_d.x = 0 \wedge \pi_d.y = 0 \quad \vee \quad \pi_d.x = 1 \wedge \pi_d.y = 1)$ – Henceforth, a stronger version of the DATA class invariant holds. This safety property is true because the routine $\pi_d.stop$ is never invoked in Zero-One.

Specifications such as [S1] and [S2] are valid iff they hold in all executions of Zero-One model. The invariant `inv_data` for class DATA (Fig. 1) can be checked either statically by the theorem prover or by run-time assertion checking. However, class ROOT (Fig. 4) never sets in motion any subsystem that triggers

[3] If the postcondition is divided into individual separate clauses, they may be checked and released separately.

routine `DATA.stop`. This is easy to see by inspection in our simple system. However, routine `stop` could occur in much bigger systems or be invoked within one of those hard to read structures such as an alternative within a loop where it is hard to decide whether it actually happens or not. If `stop` never occurs, we should actually be able to prove a stronger system invariant than `inv_data` given by:

$$\Box(x = 0 \wedge y = 0 \ \vee \ x = 1 \wedge y = 1) \tag{2}$$

Formal Verification via a theorem prover of the type discussed earlier is unable to prove this stronger property as the [CR1] rule must hold for all routines in DATA (including `stop`).

To prove properties such as S1 and S2, we note that (1) will be needed. (1) is a good rule for explaining the semantics of Scoop to compiler writers. Nevertheless, (1) as a reasoning rule is insufficient for system properties such as S1 and S2. The problem is that the postcondition of the `do_one` routine is not sufficiently projected into the future so that when execution returns to line 16, we can make use of it to argue that eventually the data will be set to zero, which would allow for the precondition of the `one` routine to be re-enabled. This will require global reasoning as discussed in [17, 16].

Testing

If we are allowed to change the code (e.g. by adding new subsystems) then runtime Assertion Testing could be used. We could allow a high priority subsystem to constantly test property (2). However, changing the implementation code is not recommended. What we need is a method to test system properties of concurrent systems without changing the code. How shall we do this?

To check system properties beyond the ones that the theorem prover for contracts can handle, we could rely on model checking and theorem proving techniques for fair transition systems. For example, we could envisage using the SPIN tool [10] or other such efficient state exploration tools.

As discussed in the introduction, formal method tools for software have steadily improved. However, they still do not fully scale up to systems of realistic size, and testing is still required for the foreseeable future. In the next section we discuss testing methods that are able to deal with Scoop programs of arbitrary size.

6 Testing, Reduced Models and SpecExplorer

In this section we show how to test for system properties beyond contracts. Essential to the method is the idea of a reduced model M_r corresponding to an original model M. The reduced model can stand in place of M for certain properties provided that $M_r \sim M$, i.e. M_r is behaviourally equivalent to M on a set of observable variables $\mathcal{O} \subset V_M \cap V_{M_r}$ which is in the intersection of the

variables set of M and M_r. The definition of behavioural equivalence is defined in [12] (page 45-47). A stronger relation – congruence of statements – is also provided in [12]. We can strengthen the notion of behavioural equivalence to also include in the observable set not only variables, but transitions (associated with feature calls) as well. Likewise, we could check for behavioural equivalence with respect to properties rather than variables.

If we want to pursue fully formal Verification, we could proceed as follows. Given a full model M for a Scoop program and a temporal logic specification S, automatically construct an appropriate reduced model M_r so that $M_r \sim M$ for the specification S (this construction could be done via abstract interpretation [8] if possible). We may then apply our analysis methods (e.g. model checking) to the reduced model M_r with a greater chance of not running into the problem of combinatorial explosion of states.

As already mentioned, these methods have been used on quite large programs, but there are still problems dealing with data and threads. Until such time that fully formal methods are capable of scaling up to deal with large programs automatically, it is still appropriate to look for Assertion Testing methods for system properties beyond contracts (see previous section).

6.1 SpecExplorer

Model-based testing is one of the most promising approaches for addressing these deficits [5]. In this paper we explore testing Scoop programs via SpecExplorer[4]. SpecExplorer is a software development tool for advanced model-based specification and conformance testing and is now used on a daily basis by Microsoft product groups for testing operating system components and .NET framework components [5]. The description below is taken from [5] and the tool website. The tool can be used to test reactive, object-oriented software systems. The inputs and outputs of such systems can be abstractly viewed as parameterized action labels, that is, as invocations of methods with dynamically created object instances and other complex data structures as parameters and return values. Thus, inputs and outputs are more than just atomic data-type values, like integers.

From the tester's perspective, the system under test is controlled by invoking methods on objects and other runtime values and monitored by observing invocations of other methods. As explained in detail in [5], this is similar to the invocation and call back and event processing metaphors familiar to most programmers. The outputs of reactive systems may be unsolicited, for example, as in the case of event notifications.

The core idea is that the developer encodes the system's intended behaviour (its specification) in machine-executable form (as a "model program"). The model program typically does much less than the implementation; it does just enough to capture the relevant states of the system and shows the constraints that a correct implementation must follow. The goal is to specify from a chosen

[4] http://research.microsoft.com/SpecExplorer

viewpoint what the system must do, what it may do and what it must not do. It can be used to explore the possible runs of the specification-program as a way to systematically generate test suites.

Discrepancies between actual and expected results are called conformance failures and may indicate a variety of problems. An implementation bug is a code defect in the implementation under test. A modeling error is a code defect in the model program itself. A specification error is a mistake or ambiguity in the system's specification (in other words, a misrepresentation of the intended system behaviour). A design error is a logical inconsistency in the system's intended behaviour.

SpecExplorer consists of an explicit-state model explorer, which allows the user to search the (possibly infinite) space of all possible sequences of method invocations that do not violate the pre/postconditions and invariants of the system's contracts and are relevant to a user-specified set of test properties. The tool has a traversal engine, which unwinds the resulting finite state machine to produce behavioural tests that cover all explored transitions. A binding mechanism allows users to associate actions of the model with methods of an implementation written in .NET languages.

6.2 Method for Testing Scoop Programs

We assume that we are provided with a Scoop program P and a system specification S. The specification does not need to be a formal temporal logic property. It may be a UML style scenario, provided we have a precise idea of what it is. For the sake of concreteness, we let P be Zero-One and the specification S is the strong system invariant [S2] of the previous section.

As stated in [5], reactive systems are inherently nondeterministic. No single agent (component, thread, network node, etc.) controls all state transitions. Network delay, thread scheduling and other external factors can influence the system behaviour. SpecExplorer handles nondeterminism by distinguishing between *controllable* actions invoked by the tester and *observable* actions that are outside of the tester's control.

We use the terms "input" and "output" relative to the system to be tested P. The terms "observable" and "controllable" will be used with respect to the inputs and outputs (respectively) of a model M_P that is used to test P for S. We proceed as follows:

1. We are provided with a Scoop program P and a system specification S. We want to know if P satisfies S, i.e. do all executions of P satisfy S? Since we are dealing with Testing and not Verification, our method will be to run a number of test executions to show that P satisfies S.
2. Use Scoopli to convert P to a .NET component.
3. Use SpecExplorer to manually construct a reduced model M_P of P. The reduced model for Zero-One is shown in Fig. 7. SpecExplorer conveniently and automatically draws the state transition graph shown in Fig. 6. We note that SpecExplorer models are close to the fair transition systems as outlined

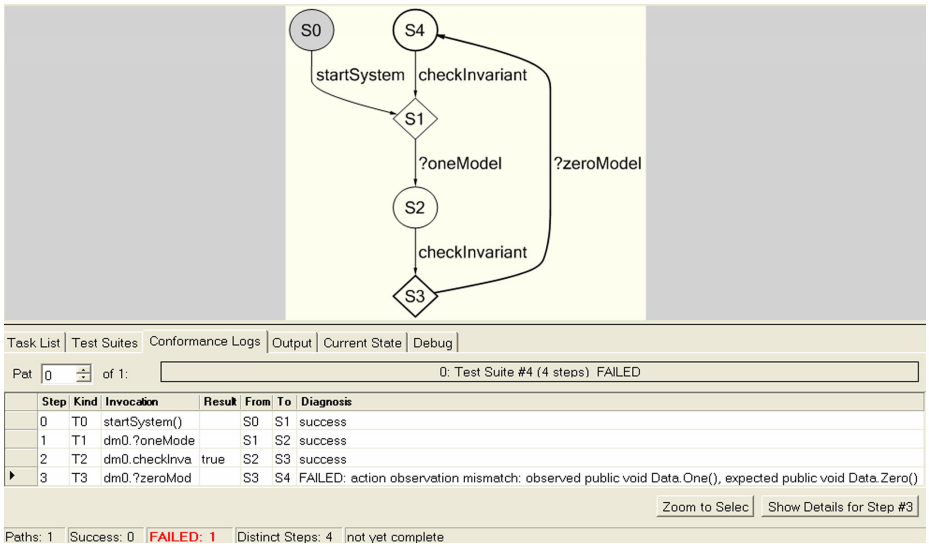


Fig. 6. SpecExplorer discovers a bug

in the previous section. The system specifications S1 and S2 of the previous section provide guidance for constructing the reduced model, e.g. that zero and one must alternate (S1), while preserving the invariant (S2).

4. We need to check behavioural equivalence, i.e. in some sense we would like to show that $M_P \sim P$. Although we cannot do this formally, we can test for behavioural equivalence using SpecExplorer's binding mechanism and graph exploration algorithms.
5. For P given by Zero-One, we bind the actions `zeroModel` and `oneModel` in Fig. 7 to the routines `DATA.zero` and `DATA.one` (respectively) as observable actions. We bind `checkInvariant` in the model to the `check_invariant` query in the ROOT class as a controllable action. The addition of this side-effect free query to ROOT is the only change that must be made to P. This query is used to check for the stronger system invariant [S2].
6. Let SpecExplorer automatically generate test cases to explore the model, run the tests and check for conformance.

The model M is written in the SpecExplorer modelling language as shown in Fig. 7 which is very close in concept to the fair transition systems (reduced or full) described in the previous section. After effecting the bindings the tool automatically generates the state exploration graph in Fig. 6. Actions (or transitions) in the model may have pre/postconditions and invariants. In the model, actions may be declared *observable* or *controllable*.

The model actions `zeroModel` and `oneModel` are declared observable and bound to `DATA.zero` and `DATA.one` respectively. This means that these model

actions are triggered whenever routines `zero` and `one` occur in the system under test P.

The model action `checkInvariant` is declared controllable and is bound to a new side-effect free query `check_invariant` in ROOT which returns true precisely when the stronger system invariant (2) holds. The round states S2 and S4 in Fig. 6 represent controllable states. When these states are reached, the controllable model action is taken thus triggering the occurrence of the bound routine in P (in this case `checkInvariant`).

```
bool systemStarted;
DataModel sharedData;

public void startSystem() requires systemStarted == false; {
    systemStarted = true;
    sharedData = createData();
}

public DataModel createData() requires systemStarted == true; {
    return (new DataModel());
}

int v = 0;

public class DataModel {
    public DataModel()
    {v = 0;}

    public void zeroModel()
    requires v == 2;
    { v = 3;}

    public void oneModel()
    requires v == 0;
    {v = 1;}

    public bool checkInvariant()
    requires v == 1 || v == 3;
    {
        if (v == 1) v = 2; else v = 0;
        return true;
    }
}
```

Fig. 7. SpecExplorer model

The model describes a system in which `zero` and `one` must alternate. If the preconditions of `zero` and `one` in DATA are removed, then the SpecExplorer

model will detect such a failure as shown by the execution to the state FAILED at the bottom of Fig. 6. This is because we expect to observe `one` and instead we saw `zero`. With the preconditions re-inserted and with the revised code fixes to ONE (section 2), the model suitably extended with a timeout will detect that the alternations occur only 1000 times and hence property [S1] of the previous section will be shown not to hold. Obviously, if we had a system that alternated for an infinite amount of time, we would be able to check [S1] for that system only for a limited period. However since [S1] failed in a finite period, the model was able to detect this.

The model is also able to show that the stronger invariant specified by [S2] holds due to the fact that `checkInvariant` is a controllable action which is invoked after each observation of the system under test.

The model can detect the deadlock failures [F1] and [F2] in section 2, which would occur if one of the DATA routines did not terminate hence not releasing the lock or if the `stop` routine is invoked. This is done by activating timeouts in the model. Failures [F3] and [F4] are detected by SpecExplorer because the system under test generates signals that are not expected in the model.

7 Conclusion

Design by contract can be appropriately extended to concurrent languages such as Scoop and Spec#. In this paper, we have compared Scoop and Spec# and shown that a contracting methodology is helpful for detecting bugs in concurrent programs. We showed how theorem provers verify that the contracts are satisfied and can be used to help detect bugs statically at compile time. Assertion Testing at run time can also detect many errors. Scoop, in particular, helps the designer avoid certain classes of race conditions by enforcing atomicity at the level of feature calls.

However, we have also shown that contracts cannot be used to detect many system level properties. We have provided the outline of a method to model Scoop programs as fair transition systems. This also allows us to describe system properties in temporal logic. These models could be used to verify Scoop programs using emerging model checking tools. Nevertheless, we expect these tools to work on systems of moderate size or on small critical components only, for the foreseeable future. Therefore, we also present an Assertion Testing methodology for Scoop programs that will scale up to programs of any size using the SpecExplorer tool.

In future work, we hope to explore better mathematical models for Scoop semantics. We are also working on equipping Scoop with powerful theorem proving tools that can be used statically to verify the contracts. We also hope to investigate the use of abstract interpretation to automatically generate SpecExplorer reduced models that are safe with respect to classes of system properties.

Acknowledgements

We thank Wolfram Schulte, Rustan Leino and Wolfgang Grieskamp of Microsoft Research for help with SpecExplorer and Spec#. We also thank Bart Jacobs for help with Spec# on the newsgroup. We gratefully acknowledge the useful feedback from the referees. This work was conducted under an NSERC Discovery grant.

References

1. Mike Barnett, Robert DeLine, Manuel Fahndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004. Verification of object-oriented programs with invariants.
2. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, volume LNCS 3362. Springer Verlag, 2004.
3. Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Meteor: A Successful Application of B in a Large Project. volume LNCS 1708, pages 369–387. Springer-Verlag, 1999. Meteor: A Successful Application of B in a Large Project.
4. L. C. Briand, Y. Labiche, and H. Sun. Investigating the use of analysis contracts to support fault isolation in object oriented code. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 70–80, New York, NY, USA, 2002. ACM Press.
5. Colin Campbell, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. In *Microsoft Research Technical Report (MSR-TR-2005-59)*. 2005. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer.
6. M. Compton. SCOOP: an Investigation of Concurrency in Eiffel. Master's thesis, Department of Computer Science, The Australian National University, 2000.
7. Oleksandr Fuks, Jonathan S. Ostroff, and Richard F. Paige. SECG: The SCOOP-to-Eiffel Code Generator. *JOT Journal of Object Technology*, 11(3), 2004. SECG: The SCOOP-to-Eiffel Code Generator.
8. Arie Gurfinkel, Ou Wei, and Marsha Chechik. Systematic Construction of Abstractions for Model-Checking. VMCAI'06, 2006. Systematic Construction of Abstractions for Model-Checking.
9. K. Havelund and T. Pressburger. Model checking Java programs using Java pathfinder. *Software Tools for Technology Transfer (STTT)*, 2(4):72–84, 2000.
10. Gerard Holzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.
11. Bart Jacobs, K. Rustan M. Leino, Frank Piessens, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, pages 137–146. IEEE, 2005.
12. Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995. Temporal Verification of Reactive Systems: Safety.
13. Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997.

14. Bertrand Meyer. The dependent delegate dilemma. volume 195 of *NATO Science Series, II: Mathematics and Physics and Chemistry*. Springer-Verlag, June 2005.
15. Piotr Nienaltowski. Efficient data race and deadlock prevention in concurrent object-oriented programs. In *Doctoral Symposium, OOPSLA 2004 Companion*, pages 56–57, 2004.
16. Piotr Nienaltowski, B. Meyer, and J.S. Ostroff. Reasoning about concurrent object-oriented programs. 2006 (to be submitted).
17. Piotr Nienaltowski and Bertrand Meyer. Contracts for concurrency. In *First International Symposium on Concurrency, Real-Time and Distribution in Eiffel-like Languages (CORDIE'06)*. Artist2 Workshop at the University of York, UK, 2006.
18. Jonathan S. Ostroff, Chen wei Wang, Eric Kerfoot, and Faraz Ahmadi Torshizi. Automated model-based verification of object-oriented code. Technical Report CS-2006-05, York University, Toronto, 2006.
19. K. Rustan, M. Leino, Greg Nelson, and James B. Saxe. ESC/Java User's Manual. Technical report, 2000. http://research.compaq.com/SRC/esc/papers.html.
20. Faraz Ahmadi Torshizi and Jonathan S. Ostroff. ESpec – a Tool for Agile Development via Early Testable Specifications. Technical Report CS-2006-04, York University, Toronto, 2006.
21. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering Journal*, 10, 2003.

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].

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

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.

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

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

$$P(x) \Longleftrightarrow \Box P(x) \Longleftrightarrow \Diamond P(x) \tag{1.4}$$

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.

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

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 $INV \wedge \forall_i \Diamond Post^j_r$ we obtain $INV \wedge \forall_i Post^j_r$ 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

$$\forall_i \, (\, \textbf{true} \ \wedge \ (\textbf{false} \ \mathcal{U} \ Post^j_r \, [a/x]))$$

and, using the property of the temporal operator *until*, to $\forall_i Post^j_r \, [a/x]$ 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 postconditions 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.

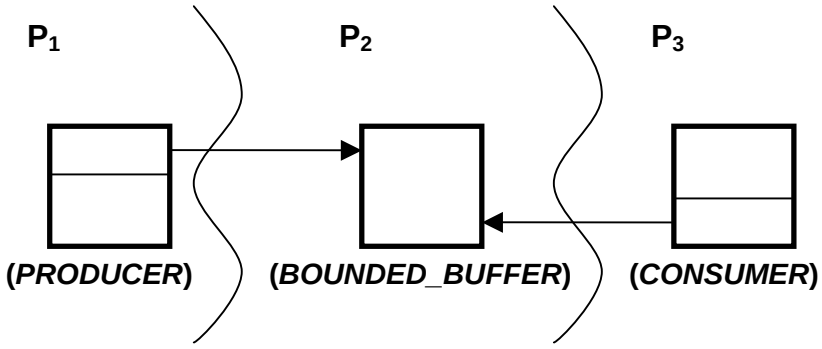


Figure 5. Producer-consumer scenario.

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.

Eiflex: Why we didn't use SCOOP

Emmanuel Bouyer, Gordon Jones

Directors and sole employees of Eifflex Ltd. - http://www.eiflex.com
emmanuel.bouyer@eiflex.com
gordon.jones@eiflex.com

Abstract. This short position paper highlights why the authors of Eiflex reliable components rejected SCOOP in the evolution of the distributed Eiflex product from a single-threaded multi-process implementation using CORBA to a multi-threaded multi-process platform.

1 Introduction

First we must acknowledge that SCOOP is something we admire immensely, and which we possibly do not fully comprehend. We would hope these comments can be of use in its evolution.

The Eiflex reliable distributed middleware is a fault tolerant "glueware bus" product written in Eiffel [Eiflex]. The first, single threaded, version of Eiflex made use of CORBA for its distribution aspects. A later multithreaded version discarded CORBA and SCOOP was considered as a replacement. However SCOOP was also dismissed, and a bespoke distribution system produced instead.

Of course one of the prime reasons SCOOP was dismissed was because at the time (2002) there was no implementation, and even now, what there is, is still a research project rather than a viable offering for production use. However had SCOOP existed, we would still have not used it. There was at least one situation where we believe we would have been unable to avoid deadlock, and others where we have reservations about its practicality.

This paper discusses these issues. These may not be entirely valid criticisms of the latest incarnation of SCOOP. However they represent our perceptions based primarily on the description of SCOOP in chapter 30 of Object Orient Software Construction Edition 2 [OOSC]. If these perceptions are a barrier for us, they may also be a barrier for other potential users.

2 Deadlock on callback

The following (incomplete and contract free) code is a frivolous example of a double dispatch pattern. We sometimes use such patterns in Eiflex, and our understanding of SCOOP is that they would result in deadlock. We do propose a possible enhancement to SCOOP that would overcome this issue.

```
class KITCHEN
feature
   prepare_steak is
      do next_plate := steak_and_chips end
   prepare_greens is
      do next_plate := broccoli_au_gratin end
   next_plate: MEAL
end

class WAITER
feature
   serve_2_diners (
         a_kitchen: KITCHEN
         a_diner, another_diner: DINER) is
      do
         a_diner.choose_meal (a_kitchen)
         a_diner.eat (a_kitchen.next_plate)
         another_diner.choose_meal (a_kitchen)
         another_diner.eat (a_kitchen.next_plate)
      end
end

class VEGETARIAN inherit DINER
feature
   choose_meal (a_kitchen: KITCHEN) is
      do a_kitchen.prepare_greens end
end

class CARNIVORE inherit DINER
feature
   choose_meal (a_kitchen: KITCHEN) is
      do a_kitchen.prepare_steak end
end

class RESTAURANT
feature
   make is
      do
         Waiter.serve_2_diners (
            Kitchen, Bertrand, Gordon)
      end
end
```

As a single threaded RESTAURANT program, Bertrand (an instance of VEGETARIAN) gets his broccoli from the Waiter via the Kitchen, and Gordon (an instance of CARNIVORE) gets his steak, and everyone is relatively happy. OK Bertrand is probably a bit unhappy that Gordon is not a vegetarian, but religion, politics, and eating habits are not things to be discussed at such times.

Now let us distribute this program and make KITCHEN, and DINER "separate" classes with the Kitchen (and restaurant and waiter) in one processor and the 2 diners in another. *waiter.serve_2_diners* reserves the kitchen and the 2 diners because they are separate arguments. The waiter first calls Bertrand to chose his meal, and since it is a procedure call, because of "wait by necessity", the waiter should be able to proceed to ask the kitchen for the next_plate to serve, and deliver to Bertrand. However there is no guarantee that Bertrand has yet told the kitchen what meal he wants (*VEGETARIAN.choose_meal*), in fact before Bertrand can choose his meal, he too must reserve the kitchen (it's a separate argument to choose_meal) to order his greens, but the kitchen is still reserved to the waiter who then presumably attempts to deliver the previous diner's plate to Bertrand. From what we understand of SCOOP we have two problems here; one Bertrand deadlocks, and two the waiter gets the wrong meal. From what we have read, we cannot determine whether either of these problems would be detected or reported.

This somewhat frivolous example of callback to an already reserved object may not be common across a distributed call sequence, but never the less they are sufficiently common in Eiflex that it was a barrier.

If we are right about the above deadlock, then we do have a suggestion as to how the SCOOP model might be enhanced to allow the distributed code to continue to work. When the *waiter.serve_2_diners* routine is entered, the waiter is holding the conch on the kitchen. When the waiter calls *Bertrand.choose_meal* passing the kitchen, the conch for the kitchen can be passed along with the call. The waiter can now no longer call the kitchen until Bertrand releases the conch. In that way there is no deadlock, and the waiter will give the correct food to the 2 diners.

There is some suggestion that SCOOP has evolved in this direction already, but it had not from the OOSC [OOSC]book.

3 Scalability of the reservation mechanism

Our next major concern was whether the reservation mechanism would scale adequately, and whether it might even undermine the achievement of the parallelism that SCOOP is trying to facilitate.

The reservation mechanism exists to enable the Command Query Separation aspect of the Eiffel method to continue to work across separate calls. Multiple objects in multiple processors are reserved (locked) for the life of the calling routine, which could be an extended period. This is to enable the routine to execute multiple commands and queries with safety. If there is contention for the services of popular resources, and they are reserved but not active for long periods, then they are unavailable to service other requests, introducing delays that could undermine throughput considerably. Also the reserve and backoff negotiation during this locked period will itself potentially generate a lot of wasted network activity.

In the enhanced version of the somewhat trivial example above, there is very little parallelism going on since the conch on the kitchen can only be held by one object at a time, and although the waiter may be released due to wait by necessity, he is quickly held up because he no longer has the reservation on the kitchen until the diner has

released it. This example is perhaps too trivial to build a strong case around, but we are sure that programmers will have to be very careful to avoid bottlenecks.

In the high throughput situations that Eiflex is targeted at, bottlenecks have to be identified and eliminated wherever they arise. The main user of Eiflex employs multi CPU SMP based hardware to maximize the potential for thread parallelism. If a badly written client reserves a number of separate target objects for extended periods, then the resulting contention would prevent the multiple CPUs from being fully utilized. In our Eiflex middleware we encourage the use of asynchronous pipelining precisely to minimize the potential for exclusion.

4 Once default

By the time this paper is presented this particular point may have been resolved, but at the time of writing, the ECMA/ISO [ECMA] standard for Eiffel, and the next version of Eiffel the Language [ETL3], both suggest the default for “once” be process wide. This would be a disaster for SCOOP, for our own Eiflex middleware, and for anyone else writing multithreaded Eiffel programs.

Most existing Eiffel libraries are not threadsafe. If their “once” (singleton) data became shared between “separate” objects in separate threads in parallel (without the control of a mutex), the libraries would be unusable in a multithreaded program – the reverse of what OO reuse is about. If the default for “once” were thread, then the libraries would remain usable, as long as SCOOP style separate, or Eiflex style channels, or windows apartment model (all of which are very similar in style) were exploited to keep the instances cleanly separated.

Actually we believe that “process once” and SCOOP are incompatible, unless the traitors mechanism is expanded to prevent the creation of the Result being non separate at compile time.

Eiflex and Eiffel Software had agreed in 2002 to use thread as the default for “once”, and we hope the ECMA body and Language reference will change the specification.

We note that Michael James Compton in his thesis paper [MJC] on SCOOP comes to the same conclusion here.

One of the (anonymous) referees of the first version of this paper pointed out to us that SCOOP might just benefit from a “system” wide “once”.

5 Wait conditions

Bertrand Meyer leads the reader of the SCOOP chapter in OOSC [OOSC] seductively from pre condition contracts on non separate routines to wait conditions when the routine is on a separate object. The point about seduction is that at the time the subject acquiesces, but sometimes, on later reflection feels uncomfortable about the path taken. We feel that this is one of those occasions.

Pre-conditions are contracts. Clients of routines are obliged to conform to them before attempting to call a routine. They an invaluable part of the Eiffel method, and

are used extensively during development and testing to check the correctness of the software. However at production time you take them out – or rather the compilation / runtime system does that for you. This is primarily for performance reasons.

Clients cannot guarantee that wait conditions are true prior to calling a separate routine. The distributed nature of the interaction and the "service" nature of the target means the client cannot be in full control. Also the compiled code must retain the wait condition checks even in production to ensure the target service is not abused by being reserved too early.

These differences seem sufficient in our minds to suggest that they are different concepts. We suspect that they cannot co-exist. That is you either have wait conditions or you have pre conditions, therefore one might argue it is redundant to insist on different syntax. However our belief is that a common syntax for 2 different concepts can introduce confusion in the mind of the software reader (and writer for that matter).

6 Separate object factories

The proposed concurrency control file does not offer the precision that we would have needed for the location of newly created separate objects. The proposal seems to supply an extremely arbitrary way of sharing out the creation of separate objects. Even to the extent that a program that works one day might deadlock another due to operating system scheduling causing separate objects to be put into different processors on different runs.

It is our view that SCOOP needs to make the processor in which a separate object is created more explicitly available to the programmer. And in addition the programmer needs to be given the ability to create new processors. Eiflex provides "thread factories" in each process (cf. processor factories) and "channel factories" in each thread (cf. separate object factories) precisely to give the programmer such control.

We believe that a SCOOP system similarly needs a processor factory, and a separate object factory in each processor. This is perhaps not a language extension but rather an extension to the base kernel standard when SCOOP is in use.

7 Conclusion

The apparent simplicity of the SCOOP language extension masks a new set of runtime complexities, some of which are discussed above. We hope that these criticisms can be addressed, particularly since we believe that others will want the same issues resolved before SCOOP can be used in a practical real world mission critical situation.

References

[OOSC] : Object-Oriented Software Construction, Second Edition, Chapter 30. 1997. Author Bertrand Meyer. Published by Prentice Hall, ISBN 0-13-629155-4.
[ETL3] : Eiffel the Language – third edition. Work in progress accessed via Bertrand Meyer's home page - http://se.ethz.ch/~meyer
[ECMA] : Eiffel language standard. 2005 - http://www.ecma-international.org/publications/standards/Ecma-367.htm
[MJC] : Michael James Compton thesis paper - http://cs.anu.edu.au/~Richard.Walker/eiffel/scoop/mc-thesis.pdf

A Critique of SCOOP

Phillip J. Brooke[1] and Richard F. Paige[2]

[1] School of Computing, University of Teesside
Middlesbrough, TS1 3BA, U.K. P.J.Brooke@tees.ac.uk
[2] Department of Computer Science, University of York
Heslington, York, YO10 5DD, U.K. paige@cs.york.ac.uk

Abstract. The Simple Concurrent Object-Oriented Programming (SCOOP) is the leading proposed mechanism for introducing concurrency to Eiffel. We summarise our position on the status of SCOOP, and on the open issues and research questions that should be addressed.

1 Introduction

The Simple Concurrent Object-Oriented Programming (SCOOP) [7, 8] mechanism is proposed as a way to introduce inter-object concurrency into the Eiffel programming language [5, 7]. SCOOP extends the Eiffel language by adding one keyword, **separate**, which can be applied to classes, entities, and formal routine arguments. Application of **separate** to a class indicates that objects of that class execute in their own (conceptual) 'thread' of control. Further, **separate** applied to entities means that the attached object is (potentially) running on a different subsystem, and when applied to a parameters of a feature call, indicates that the attached object should be reserved (locked) for the duration of that call.

2 Critique

Previous attempts to implement SCOOP [4, 6, 9] have proved difficult; we suggest that the major reason is that the SCOOP mechanism and its underlying semantics are both complex to understand and thus difficult to implement in a compiler and run-time environment. The complexities inherent in the interactions between the language and the implicit, underlying run-time system are potentially confusing.

Given our understanding of SCOOP, based on [7, 8], the mechanism suffers from under-specification and a number of potentially undesirable behaviours.

2.1 Reservations and call chains

The execution of a feature call requires the reservation (locking) of each of the separate objects given in the parameters of that call. One problem concerns chains of such calls, each of which reserves the same separate object. If $a.f$, i.e., feature f of the object attached to entity a, wishes to call feature $b.g$, then the current model of SCOOP requires that

1. a must have a reservation on b before it can call g in b.
2. b needs to obtain reservations on each (separate) argument in the call before it can execute g.

Now suppose that $b.g$ is a function call and that $a.f$ requires the result of that call before it can proceed past a certain point. Moreover, a also holds the reservation on a further separate object s and s is one of the arguments to $b.g$. In this case, deadlock will result:

- $a.f$ cannot make progress because it requires $b.g$ to return its result; but
- $b.g$ cannot start because $a.f$ has reserved s; and
- $a.f$ will not release s until it has finished.

This is clearly undesirable, and is a serious problem since object-oriented programming often includes chains of calls passing on the same argument.

A different paper [3] proposes that this problem is solved by allowing features to 'pass on' their reservations to calls they themselves make; while they have passed on their reservation, they cannot make progress that requires it.

The first author identified this problem in October 2004 and has been advocating the proposed solution since then. One issue related to this solution is that it may make reasoning about concurrent systems more difficult, while at the same time increasing the potential for concurrency.

2.2 Release of reservations

When should a reservation be released? Should the reservation be released

- as soon as the last reference to the reserved object has been processed,
- as soon as the end of the reserving feature is reached, or
- only when all calls made by the feature have also finished?

The first option requires the compiler to determine this from the text of the program; the second option is the most obvious, but may have further subtle effects on mutual exclusion and ordering of execution; the final option substantially reduces the amount of parallelism in the system.

2.3 Parallelism, subsystems and reservations

The SCOOP mechanism has potential for very high parallelism: each object can be viewed as active in its own right. However, two factors can potentially reduce this parallelism:

- Subsystems group 'relatively local' objects together by providing 'handling' for groups of objects on the same 'processor'. Thus objects in a subsystem share a common work queue and cannot execute feature calls simultaneously. This could potentially reduce parallelism, but the programmer could increase parallelism by increasing the use of **separate** variables in their code.

– OO systems typically pass on arguments to subsequent calls: real systems may reserve large amounts of objects for considerable time. Such objects are then denied to other features, again reducing potential parallelism.

The latter aspect might be partially solved by allowing multiple readers simultaneous access (or some partitioning of classes[3]).

2.4 Rescheduling blocked preconditions as wait conditions

Preconditions in SCOOP are treated as wait conditions. A call can only proceed when its preconditions evaluate to true. However, nothing in SCOOP describes how long can or should pass between the initial failure and subsequent re-evaluation. If several calls are blocked on the same condition, [7, p.996] suggests a default first-in-first-out policy for identifying the order in which calls proceed, with an option for the use of library mechanisms to override this default. The behaviour of the mechanism, particularly in complicated systems, is as-yet undetermined.

2.5 Preconditions *and* wait conditions?

The argument in Meyer's text [7], which treats sequential preconditions as (concurrent) wait conditions in SCOOP, is compelling. This is typically illustrated using buffer examples. However, we have recently come to the view that we may still require preconditions in the concurrent environment, e.g., where the precondition is a safety check and will never become true due to another object's behaviour. One issue for discussion is whether these safety checks should be treated as contracts (i.e., to be checked and relied on by clients) or as conditions to be checked within a routine call.

2.6 Queued calls and priorities

Calls to separate objects are not executed immediately: instead, they are queued on the object's handler (a subsystem). This gives the potential for many calls to be enqueued on one handler.

There is no obvious way to embed priorities into this model without causing priority inversion, or breaking mutual exclusion and causing races.

2.7 separate complications

SCOOP is inherently complicated: it has multiple layers in its semantic model, since objects are grouped into *sequential* subsystems. Thus an object in one subsystem calling another object in that subsystem results in the normal sequential call semantics.

[3] Another idea we are pursuing concerns the use of a keyword **allow** (related to **only**) to enable partitioning of classes into disjoint parts.

Additionally, the language rules require extra *separateness consistency rules* to prevent *traitors*: the assignment of a separate object to a non-separate entity. However, the assignment of a local object (i.e., one on the same subsystem) to a separate entity results in synchronous processing when asynchronous processing might have been expected.

Finally, the use of the keyword **separate** for indicating mutual exclusion (via the formal argument list) results in difficulty passing separate reference: we do not always need to reserve the object referenced by an entity in the arguments (it might be intended to be passed to a later call).

These problems may disappear if one treats synchrony as a special case of asynchrony. However, this may again make reasoning about concurrency more difficult; experiments on real systems need to be carried out to determine how much of a difficulty this will be in practice.

2.8 Exceptions and real-time

The discussion of SCOOP in [7, 8] does not consider important real-time and exception handling issues, notably:

Exceptions in asynchronous calls are not described in [7, 8], even though a mechanism for demanding urgent service using exceptions exists ('duels' in Meyer's text).
Where should the exception be delivered when the caller has already terminated? Or do all callers block? — reducing parallelism once again.
A discussion on exceptions in real-time Eiffel is considered in [1], and additional information will be presented at the CORDIE workshop [2].

Real-time constructs are also missing. This could be examined both with and without SCOOP.

Interrupts from external processes are lacking: this is related to both exceptions and real-time behaviours.

2.9 Implementation matters

An additional source of exceptions could arise from failure of the underlying communication system, e.g., network failure between several processing nodes. Fault-tolerance, resilience and redundancy are not currently addressed. A failure in the communication system needs to be propogated to the caller.

A real implementation for large systems will have to handle the concurrent reservation of objects. Since a failed attempt to reserve all the objects needed for a call results in all being released prior to a later re-attempt, we may find races on the collection of reservations.

Simplistic algorithms for termination are easy to describe. In the sequential case, the execution is started with the creation procedure of the root object. When that routine finishes, the whole execution finishes.

In SCOOP, this root creation procedure can start off a series of separate objects. So the execution of the system as a whole only ceases

– when the root creation procedure has finished; and
– every separate object (including the root object) is quiescent.

Essentially, every object in the system is not executing and each subsystem has an empty job queue. If nothing is executing, then nothing can queue any calls. This has to be determined completely at one point in time across the whole system to prevent errors. Doing so efficiently may be difficult.

Deadlock is similar, although it requires some way to detect that all executing calls are blocked waiting for results or reservations.

It is unclear how the infrastructure for modestly large systems should be implemented. A system that contains many thousands of objects running on a thousands of subsystems may have to run on a machine with only a few thousand UNIX processes or threads available. This means that the subsystems themselves must be multiprogrammed onto these relatively scarce resources.

Scalability in general is an open question.

3 Conclusion

We have identified the areas that we consider problematic in SCOOP. However, we consider that this model is worth effort to fix: a higher-level model suited for OO programming would be far superior to the current low-level threads-and-mutexes approaches currently used.

To address the issues we describe, the current approaches of building preprocessor implementations can be usefully augmented by other compilers, a battery of test cases, and formal models.

Acknowledgements

We thank the referees and the ETHZ SCOOP researchers for their helpful comments and discussions.

References

1. V. Arslan, P. Eugster, and P. Nienaltowski. Modeling embedded real-time applications with objects and events. *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006)*, April 2006, San Jose, USA.
2. V. Arslan and B. Meyer. Asynchronous exceptions in concurrent object-oriented programming. To appear in *Proc. CORDIE 2006*, York, UK, July 2006,
3. P.J. Brooke, R.F. Paige, and J.L. Jacob. A CSP model of Eiffel's SCOOP. Making changes following FACJ reviewers' comments received on 20th April. Should resubmit in one or two months., November 2005.
4. M. Compton. SCOOP: an investigation of concurrency in Eiffel. Master's thesis, Australian National University, 2000.
5. ECMA-367: Eiffel analysis, design and programming language. ECMA International, June 2005.

6. O. Fuks, J.S. Ostroff, and R.F. Paige. SECG: The SCOOP-to-Eiffel code generator. *Journal of Object Technology*, 3(10), November/December 2004.
7. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.
8. P. Nienaltowski, V. Arslan, and B. Meyer. SCOOP: Concurrent programming made easy. Draft paper from ETH Zürich, http://se.inf.ethz.ch/people/nienaltowski/papers/scoop_easy_draft.pdf, 2004.
9. P. Nienaltowski and B. Meyer. SCOOPLI implementation, 2005. http://se.inf.ethz.ch/research/scoop.html.

Asynchronous Exceptions in Concurrent Object-Oriented Programming

Volkan Arslan, Bertrand Meyer

Chair of Software Engineering, ETH Zurich
{Volkan.Arslan, Bertrand.Meyer}@inf.ethz.ch
http://se.ethz.ch

Abstract. Exceptions in concurrent object-oriented languages with asynchronous call semantics may raise a serious problem in certain situations. Since separate calls are asynchronous it might happen that the context of the enclosing routine, from which the asynchronous call was launched, has been already left and hence any exception raised by the asynchronous call can not be handled anymore by the enclosing routine. In this paper we present a practical solution for this problem, which relies on the notion of busy processors.

1 Introduction

Exceptions play an important role in programming robust software systems. According to [1] informally, an exception is an abnormal event that disrupts the execution of a system. In sequential (object-oriented) programming languages relying on Design by Contract technique, exceptions have a very clear and precise semantics. The situation changes dramatically as soon as one moves from sequential to concurrent programming, especially if the underlying concurrency model relies on so-called asynchronous calls; that is calls that are not blocking.

The rest of this paper is organized as follows: Section 2 explains shortly the semantics of exceptions in sequential object-oriented languages such as Eiffel. Section 3 first outlines shortly the concurrency model used in this paper and then describes the exact problem with exceptions in concurrent object-oriented programming and afterwards presents our proposed exception mechanism. Section 4 draws conclusions and discusses possible extensions of the proposed exception mechanism.

2 Exceptions in sequential programming

The semantics of exceptions in sequential object-oriented languages such as Eiffel relying on the Design by Contract technique is extremely easy to understand and to explain. Before we give a precise definition for the term exception, we have to give the definitions for routine success and failure.

Definition: success, failure

A routine call succeeds if it terminates its execution in a state satisfying the routine's contract. It fails if it does not succeed.

The routine's contract consists of its precondition and postcondition. Additionally the implementation of the routine can have assertion checkings through the **check** keyword. It should be noted that a class has also a class invariant which must hold after the execution of any publicly exported routine. Now we can give the definition for the term exception:

Definition: exception

An exception is a run-time event that may cause a routine call to fail.

Having given the definitions for routine success and failure, we note that a routine call will fail if and only if an exception occurs during its execution and the routine does not recover from the exception. In general there are various ways how to deal with the occurrence of exceptions, but in Eiffel the exception mechanism supports the Disciplined Exception Handling Principle:

Disciplined Exception Handling Principle

There are only two legimate responses to an exception that occurs during the execution of a routine:.

1 • **Retrying:** attempt to change the conditions that led to the exception and to execute the routine again from the start

2 • **Failure:** (also known as **organized panic**): clean up the environment, terminate the call and report failure to the caller.

The above disciplined exception handling principle is supported through the **rescue** and **retry** clauses.

```
r
        require
                precondition
        local
                ... local entity declarations
        do
                body
        ensure
                postcondition
        rescue
                rescue_clause
        end
```

A routine *r* might have a rescue clause. In case of an exception during the execution of the normal routine *body*, the execution in the body part will stop and the *rescue_clause* will be executed instead. Inside the *rescue_clause* there can be a **retry** instruction whose execution will force to re-start the routine body from the beginning without repeating the initialization of the routine. As stated above a routine such as *r* might either fail after executing the *rescue_clause* if there is no **retry** instruction or

succeed after a **retry** instruction. It should be noted that the execution of the *rescue_clause* has to establish the class invariant and additionally the precondition if the **retry** instruction is executed. If a routine fails the exception will be propagated to the caller routine, that is it to the routine which called *r*.

3 Exceptions in concurrent programming

3.1 The SCOOP model

The SCOOP model (Simple Concurrent Object-Oriented Programming) [1], [2] offers a comprehensive approach to building high-quality concurrent and distributed systems. The idea of SCOOP is to take object-oriented programming as given, in a simple and pure form based on the concepts of Design by Contract, which have proved highly successful in improving the quality of sequential programs, and extend them in a minimal way to cover concurrency and distribution. The extension indeed consists of just one keyword **separate**; the rest of the mechanism largely derives from examining the consequences of the notion of contract in a non-sequential setting. The model is applicable to many different physical setups, from multiprocessing to multithreading, network programming, Web services, highly parallel processors for scientific computation, and distributed computation. For application programmers, writing concurrent applications with SCOOP is extremely simple, not requiring the usual baggage of concurrent and multithreaded programming (semaphores, rendezvous, conditional critical regions etc.). The model takes advantage of the inherent concurrency implicit in object-oriented programming to provide programmers with a simple extension enabling them to produce concurrent applications with little more effort than sequential ones.

Processors

SCOOP uses the basic scheme of the object-oriented computation: the feature call, e.g. *x.f* (*args*), which should be understood in the following way: the client object calls feature *f* on the supplier object attached to *x*, with the argument *args*. In a sequential setting, such calls are synchronous, i.e. the client is blocked until the supplier has terminated the execution of the feature. To introduce concurrency, SCOOP allows the use of more than one processor to handle execution of features. A processor is an autonomous thread of control capable of supporting the sequential execution of instructions on one or more objects. It can be implemented by a piece of hardware (CPU), a process, a single thread in a multithreaded environment, or an application domain in Microsoft .NET, etc. If different processors are used for handling the client and the supplier objects, the feature call becomes asynchronous: the computation on the client object can move ahead without waiting for the call to terminate. Processors are the principal concept that SCOOP adds to the sequential object-oriented framework. Contrary to a sequential system, a concurrent system may have any number of processors, independently of the number of available CPUs.

Separate calls

A declaration of an entity or function, which normally appears as *x*: *X* may now also be of the form *x*: **separate** *X*. Keyword separate indicates that entity *x* is handled by a different processor, so that calls on *x* should be asynchronous and can proceed in parallel

with the rest of computation. With such a declaration, *x* becomes a separate entity. If the target of a call is a separate expression, i.e. a separate entity or an expression involving at least one separate entity, such call is referred to as separate call.

Synchronization

No special mechanism is required for a client to resynchronize with its supplier after a separate call *x.f* (*args*) has gone off in parallel. The client will wait if and only if it needs to, i.e. when it requests information on the object through a query call, as in *value* := *x.some_query*. This automatic mechanism is known as wait by necessity [3]. SCOOP ensures that the separate calls made by the client to each supplier are executed in the correct order (FIFO).

Contracts and preconditions

SCOOP relies largely on the principles of Design by Contract. In particular, it introduces a new semantics for preconditions. The semantics of preconditions is different in sequential and concurrent setting. In sequential programs, preconditions are assertions that have to be fulfilled by the client object before calling the routine of the supplier object. If one or more preconditions are not met, the contract is broken and an exception is raised in the client object. In a concurrent context, the preconditions which do not involve any separate entities (e.g *value_specified* in the example routine *store* below) keep their original semantics: they are correctness conditions.

The preconditions involving calls on separate objects (e.g. *buffer_not_full*) change their semantics:

```
store (b: separate BUFFER [G]; v: G)
            -- Store v in b.
        require
            value_specified: v /= Void
            buffer_specified: buffer /= Void
            buffer_not_full: not b.is_full
        do
            b.put (v)
        ensure
            buffer_not_empty: not b.is_empty
        end
```

They become wait conditions. If such precondition is not satisfied, it does not result in an exception raised in the client; it only causes the client to wait until the precondition is satisfied.

3.2 The problem with exceptions in asynchronous feature calls

The basic problem with exceptions in concurrent programming with asynchronous feature calls such as *x.f* (*args*) where *x* is a separate entity, is that it might happen, that the context of the enclosing routine, from which the asynchronous call was launched, has been already left and hence any exception raised by the asynchronous call can not be handled anymore by the enclosing routine. To illustrate the problem in more detail con-

sider class *X* (see /1/). Class *X* consists of the routines *f*, *g*, and *establish_invariant* which are commands and of *query* which is a query returning a boolean value.

```
class X feature /1/

f
        require
                precondition_1
        do
                ...
        ensure
                postcondition_1
        end

g
        require
                precondition_2
        do
                ...
        ensure
                postcondition_2
        end

query: BOOLEAN
        do
                ...
        end

establish_invariant
        do
                ...
        end

end
```

As a side note commands can change the state of objects, whereas queries return information about objects. Furthermore /2/ lists a class *C*1 which uses the class *X*. *C*1 is said to be a client of *X*, and *X* a supplier of *C*1.

In routine *start* in class *C*1 the call *r* (*my_x*) causes to reserve the separate object, to

```
class C1 feature /2/

start
        do
                r (my_x)
                ...
                z := z + 1
                ...
                s (my_x)
        end

my_x: separate X

z: INTEGER

r (x: separate X)
        do
                x.f
                x.g
        end

s (x: separate X)
        require
                x.query
        do
                x.f
        rescue
                x.establish_invariant
                retry
        end

end
```

which *my_x* is attached, and afterwards the asynchronous call of the feature *f* on the formal argument *x*. Since this call *x.f* is a non-blocking call the next instruction *x.g* can also be launched immediately after the first asynchronous call. The routine *r* will then terminate since there is no other instruction in *r* and one thread of program execution will continue with the next instruction *z* := *z* + 1 in routine *start* of class *C*1.

Now it can happen that one or both of the feature calls *x.f* and *x.g* fail due to an exception raised either in *f* or *g* of class *X*. Since the caller of the feature calls *x.f* and *x.g* (which is *r*) has already left the context, it is clear that *r* cannot handle anymore the exceptions propagated by *f* or *g* of class *X*. The sketched scenario above is a severe

problem in concurrent object-oriented programming. It should be noted that the above problem would not appear in the following modified routine *r* of class *C*1:

```
r (x: separate X)
        do
                x.f
                x.g
                res := x.queryf
        end
```

Since the last instruction in the routine *r* now is a query instead of a command, whose result is assigned to an entity *res*, the call *x.query* will be thanks to wait by necessity synchronous and hence the context will be not left in case of an exception.

3.3 Proposed asynchronous exception mechanism

The proposed exception mechanism relies on the notion of *busy processors*. A processor is called *busy* if an exception has been raised by any object handled by this processor. Let us illustrate the proposed solution again through a simple example. Assume the following declarations:

```
c1: separate C1

c2: separate C2

c3: separate C3
```

The class code of *C*2 (see /3/:

```
class C2 feature /3/

r (x: separate X)
        do
                x.f
        end

end
```

and *C*3 (see /4/) are similar to those of *C*1

```
class C3 feature /4/

r (x: separate X)
        do
                x.f
        end

end
```

We assume that on behalf of *c*1 the feature *r* (through the feature *start*) has been executed; similarly on behalf of *c*2 and *c*3 the feature *r* has been executed. Hence all three objects are competing for the processor *PX* of the shared separate object *x*. Assume that the processor *P*1 of *c*1 reserves or locks first the processor *P*X. This means that both the processors of *c*2 and *c*3 (*P*2 and *P*3) have to wait until the processor *PX* is free again. Now *P*1 asynchronously calls *x.f* and *x.g* and leaves the context of *r* and continues to execute the next instruction in the routine *s*, which is the assignment instruction. In the meanwhile assume that the asynchronous call *x.f* fails due to an exception. In this case the processor *PX* is declared as "busy" meaning that only objects of the processor *P*1 can access the processor *PX*. This means in particular that an object of *P*1 has to bring the state of the processor *PX* from "busy" to "normal" since *P*1 was the originator of the exception through the routine *r* of *C*1. For the processors *P*2 and *P*3 the processor *PX* will still be busy meaning that the processor *PX* is not available for them. This is similar to the case where several separate processors are competing to lock a certain processor, but only one processor can succeed and the others have to wait.

The interesting question now is what should happen when *P*1, the originator of the exception in *PX*, again accesses *PX*. In this case *P*1 should get the pending exception. In our example this will be the case when *P*1 executes *s* (*my_x*) and tries to lock *PX*. Since *PX* is in busy state, and *P*1 is the originator of this state, *P*1 will get the exception before it enters the body of the routine *s* without the need to wait until *PX* is free and without checking for the waitconditon *x.query*. *P*1 will immediately continue in the rescue clause of the routine *s*. There *P*1 has then the possibility to reestablish the invariant of the separate object attached to *x* by calling *x.establish_invariant* and then to call retry to continue the execution in the body of the routine *s*. As soon as the pending exception is handled, the state of the processor *PX* will be set from "busy" to "normal". Now when *PX* is again in "normal" state, other processors such as *P*2 and *P*3 can access *PX* as if nothing has happened.

The advantage of this solution lies in the fact that other processors such as *P*2 and *P*3 are not punished for the exception which has been not caused by any objects of *P*2 and *P*3. Since *P*1 is the originator of the exception, an object handled by *P*1 is the best suited one to resolve the problem. One major disadvantage for the processors *P*2 and *P*3 is of course the fact that in certain situations they have to wait indefinitely for the processor *PX*, if no other object of *P*1 accesses *PX* and hence brings *PX* into normal state. But again the problem can be solved by *P*2 and *P*3 by introducing timeouts. After a certain amount of waiting time, *P*2 and *P*3 can access *PX*, but then they will be confronted immediately with the pending exception.

4 Conclusions and ongoing work

We presented a simple solution for the exception mechanism in concurrent object-oriented languages relying on asynchronous calls. It should be noted that there are not many concurrent object-oriented languages relying on asynchronous calls. One previous work [4] relied on wait by rescue, meaning that whenever a routine has a rescue clause, the routine had to wait until all asynchronous calls terminated. This approach was not

very efficient from performance aspects, since all calls in such routines degenerated to synchronous calls loosing the attractiveness of concurrent programming.

We are currently in the process of integrating the proposed exceptions mechanism into our SCOOP library called SCOOPLI [5]. We are also extending our SCOOP library with a timeout mechanism and specific support for periodic and aperiodic real-time tasks [6].

References

[1] Meyer B.: *Object-Oriented Software Construction, 2nd edition*, Prentice Hall, 1997.

[2] Nienaltowski P., Arslan V.: *SCOOPLI: A library for concurrent object-oriented programming on .NET*; in 1st International Workshop on C# and .NET 2003, University of West Bohemia, Plzen, Czech Republic.

[3] Caromel D.: *Towards a Method of Object-Oriented Concurrent Programming*; in Communications of the ACM, Volume 36, Number 9, September 1993, 90-102

[4] Nenning C.: *Exception Handling in SCOOP*, Master's thesis, 2004, ETH Zurich

[5] SCOOP Library: available for download at http://se.inf.ethz.ch/research/scoop

[6] Arslan V., Eugster P., Nienaltowski P.: *Modeling Embedded Real-Time Applications with Objects and Events*; in 12th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2006, San Jose, California , United States.

Flexible locking in SCOOP

Piotr Nienaltowski

Chair of Software Engineering
Swiss Federal Institute of Technology Zurich
CH-8092 Zurich, Switzerland
piotr.nienaltowski@se.inf.ethz.ch

Abstract. The SCOOP model provides programmers with a simple extension of Eiffel that allows them to produce high-quality concurrent applications with little more effort than sequential ones. The model is simple yet powerful. Nevertheless, its access control policy is pessimistic: (1) all separate actual arguments of a feature call are locked, even if it is not necessary, and (2) at most one client object can access a given supplier object at any time. This results in increased potential for deadlocks; additionally, some interesting synchronisation scenarios cannot be implemented efficiently. This paper presents two mechanisms that increase the flexibility of locking in SCOOP: (1) a type-based mechanism to specify which arguments of a routine call should be locked, and (2) a lock-passing mechanism that allows for safe handling of callbacks and complex synchronisation scenarios that involve mutual locking of several separate objects. When combined, these two approaches greatly increase the expressive power of SCOOP and reduce the risk of deadlock.

1 Introduction

Controlling access to shared resources is one of the main problems in concurrent programming. Uncontrolled access to shared resources is very dangerous as it may lead to an inconsistent program state. In procedural programming, solutions to conflict problems involve proper synchronisation among processes based on the concept of critical section – a process requesting a shared resource has to wait for executing its critical section if another process is currently accessing the shared resource. The situation changes significantly when we deal with object-oriented computations. Explicit critical sections are not necessary because they may be encapsulated in class routines, as in the SCOOP model [1]. The most important question is how to ensure that concurrent calls to the routines of the same object do not cause deadlock and do not violate the integrity of the object (i.e. the invariant of its base class). An appropriate locking policy may be applied in order to ensure these two conditions. The SCOOP model proposes such a policy. SCOOP-based applications satisfy the safety requirements – they exhibit no data races and no invariant violations due to parallelism. Unfortunately, this comes at a very high price: all accesses to a separate supplier object must be wrapped in a routine body that represents a critical section; this results in a very coarse-grained parallelism. Also, all separate arguments of a feature call have to

be locked, even if they are never used by the feature. Additionally, a client that holds a lock on a given resource cannot relinquish it temporarily when the lock is not needed. As a result, certain scenarios, e.g. callbacks involving separate suppliers, cannot be implemented. In most cases, the amount of locking is higher than necessary. Such a pessimistic locking policy makes SCOOP-based programs more deadlock-prone.

We present two ways of relaxing the access control policy: (1) we introduce a mechanism for specifying which arguments of a routine call should be locked, and (2) we allow clients to temporarily pass on their locks to separate suppliers. We illustrate the discussion with numerous code examples.

The article is organised in the following way. Section 2 shortly describes the basic synchronisation policy of SCOOP. Section 3 describes the type-based mechanism for precise specification of formal arguments to be locked. Problems of precondition weakening and precursor calls discussed in that section are not concurrency-specific; their analysis and the proposed solution (rule 1') may be regarded as contributions to DbC in general. Section 4 introduces the lock-passing mechanism. Section 5 discusses related work. Finally, Section 6 concludes the article and describes future research directions.

The use of detachable and attached types in the context of SCOOP is part of joint work with Bertrand Meyer. The need for lock passing – as a way to clarify SCOOP semantics – was initially pointed out by Phil Brooke and later reflected in the CSP semantics for SCOOP [2] in the form of transitive locking, whereby suppliers are allowed to "snatch" a lock from their clients when necessary. Although we use the same name for our mechanism, we follow a different approach here: we require a client to pass locks explicitly; our goal is to increase the flexibility of the model while preserving the possibility to reason about the order of feature calls. The differences between both solutions are discussed in section 5.

2 SCOOP model

The SCOOP model (Simple Concurrent Object-Oriented Programming) offers a disciplined approach to building high-quality concurrent systems. The idea of SCOOP is to take object-oriented programming as given, in a simple and pure form based on the concepts of Design by Contract [3], which have proved highly successful in improving the quality of sequential programs, and extend them in a minimal way to cover concurrency and distribution. The extension consists of just one keyword **separate**; the rest of the mechanism largely derives from examining the consequences of the notion of contract in a non-sequential setting.

2.1 Processors

SCOOP uses the basic scheme of object-oriented computation: the feature call $x.f\ (a)$, which should be understood in the following way: the caller object calls

feature f on the supplier object attached to x, with the argument a. In a sequential setting, such calls are synchronous, i.e. the caller is blocked until the supplier has terminated the execution of the feature. To introduce concurrency, SCOOP allows the use of more than one *processor* to handle the execution of features. A processor is an autonomous thread of control capable of supporting the sequential execution of instructions on one or more objects. If different processors are used for handling the caller and the supplier objects, the feature call becomes asynchronous: the computation on the caller object can move ahead without waiting for the call to terminate. Processors are the principal concept that SCOOP adds to the sequential object-oriented framework. Contrary to a sequential system, a concurrent system may have any number of processors, independently of the number of available CPUs.

2.2 Separate calls

A declaration of an entity, which normally appears as x: *SOME_CLASS* may now also be of the form x: **separate** *SOME_CLASS*. Keyword **separate** indicates that entity x is handled by a (potentially) different processor, so that calls on x might be asynchronous and may proceed in parallel with the rest of computation. With such a declaration, x becomes a separate entity. If the target of a call is a separate expression – a separate entity or an expression involving at least one separate entity – such call is referred to as *separate call*.

2.3 Synchronisation

SCOOP caters for the synchronisation and communication needs of concurrent programming such as mutual exclusion, locking, and waiting by relying on Design by Contract and argument passing.

Mutual exclusion A basic rule of SCOOP says that a separate call $an\_x.f\ (a)$ (where an_x is separate) is only permitted if an_x appears as formal argument of the enclosing routine; calling a routine with such a separate argument will make the client object wait until the corresponding separate supplier object is exclusively available to the caller. So, if the client calls $r\ (x)$, where routine r is defined as

```
r (an_x: separate X)
    do
        ...
        an_x.f (a)
        ...
    end
```

the call will wait until the processor handling x is available to the client (i.e. no other client is using it). This rule provides the basic synchronisation mechanism for SCOOP. It avoids the most common mistake in concurrent programming

that consists in assuming that, when making two successive calls on a separate object, e.g.

my_stack.push (*some_value*)
...
$x :=$ *my_stack.top*

nothing may happen to the object represented by *my_stack* between the two calls. In the example above, we would expect that the object assigned to x is indeed the object denoted by *some_value* that we just pushed on *my_stack*. Unfortunately, such "sequential thinking" does not apply in a concurrent setting, since other clients may interfere with the object referred to by *my_stack* between the two calls. In SCOOP, routine bodies represent critical sections (w.r.t. to their separate arguments) – the client gets an exclusive access to all the processors that handle the separate arguments of the routine. In the example above, *my_stack* must be an argument of the enclosing routine, therefore there is no danger that another client "jumps in" and modifies the state of the supplier object between two consecutive calls issued by our client.

Condition synchronisation SCOOP provides support for condition synchronisation by giving a different semantics to preconditions in a concurrent context. Precondition clauses that involve separate calls become *wait-conditions*; the client object is forced to wait until they are satisfied. We do not discuss the condition synchronisation mechanism any further here because it is not influenced by the new access control policy; interested readers should refer to [1] for more details. In a separate article [4] we propose a generalised semantics for contracts in SCOOP that unifies the concepts of preconditions and wait-conditions.

Resynchronisation No special mechanism is required for a client object to resynchronise with its supplier after a separate call $x.f\ (a)$ has gone off in parallel. The client will wait if and only if it needs to, i.e. when it requests information on the object through a query call, as in *value* $:=$ $x.$*some_query*. This automatic mechanism is known as *wait-by-necessity* [5]. The lock-passing mechanism described in section 4 will slightly modify that policy: procedure calls that involve lock-passing will also require the client object to wait, as in the case of a query call.

3 Eliminating unnecessary locks

In this section, we take the first step towards relaxing the locking policy of SCOOP – we present a simple mechanism that allows the programmer to specify precisely which formal arguments of a routine should be locked. This allows us to eliminate the unnecessary locking – only the locks that are strictly necessary will be acquired. The mechanism relies on the concept of *detachable types* recently introduced in the Eiffel language [6]; it is fully compatible with other

object-oriented concepts such as polymorphism, inheritance, and genericity. The application of detachable types to SCOOP is a result of joint work with Bertrand Meyer; the basic idea was described in [7]. Here, we take a closer look at the mechanism, discuss its applications, and study its impact on other language features.

3.1 (Too much) locking considered harmful

Recall that SCOOP requires that all separate arguments of a routine call be locked before the call can proceed. This policy is too restrictive and it unnecessarily increases the likelihood of deadlock. Consider feature r in Figure 1. According to SCOOP, the processors that handle x, y, and z must be locked by

```
r (x: separate X; y: separate Y; z: separate Z)
    require
        some_precondition
    local
        my_y: separate Y
        my_z: separate Z
    do
        x.f  -- separate call
        my_y := y
        x.g  -- separate call
        my_z := z
        s (z)
    end
```

Fig. 1. Original feature

the client object before the body of r can be executed. Is it really necessary to lock all of them? Let's see: the body of r contains two calls on x, therefore x needs to be locked. There is no way around it – we must ensure that no other client is currently using x. On the other hand, y only appears on the right-hand side of an assignment; no calls on y are made. Similarly, z only appears as source of an assignment and as actual argument of a feature call. It seems that we only need to lock the processor that handles x; it is not necessary for y and z because the body of r does not contain any calls on them.

The eager locking applied by SCOOP might be very dangerous as it often leads to deadlocks – the more resources a client requires, the more likely it is to get in a deadlock situation. The locking policy can be easily refined to avoid these drawbacks.

3.2 Detachable types and their concurrent semantics

The *attached type* mechanism is an extension of Eiffel's type system [6]. Every type is declared either as "attached" or as "detachable"; an attached type guar-

antees that the corresponding values are never void. The default case is attached, e.g. x: X means "x is of type *attached* X". Detachable types are marked with '?', e.g. y: ?Y means "y is of type *detachable* Y". A qualified call $x.f\ (a)$ is valid only if the type of x is attached. A new validity rule allows an attachment (assignment or argument passing) from the attached version of a type to the detachable version but not the other way round (unless a check of non-voidness is performed) [7]. We can rely on the use of detachable and attached types to specify which arguments of a routine should be locked. We require that all attached formal arguments of a routine be locked. Conversely, no detachable formal arguments are locked. This is not a mere overloading of the semantics of detachable types. In fact, this rule captures the essence of call validity: a client is allowed to make a call if and only if the target is non-void and the client has exclusive access to the target's processor. We use attached annotations to satisfy both requirements. Let's apply the rule to the example in Figure 1. Now, only the processor that handles x will be locked when a call to r is executed. The processors that handle y and z will not be locked (see Figure 2). Note that the applied rule is consistent

```
r (x: separate X; y: ?separate Y; z: ?separate Z)
    local
        my_y: ?separate Y
        my_z: ?separate Z
    do
        x.f
        my_y := y
        x.g
        my_z := z
        s (z)
    end
```

Fig. 2. Redefined feature

with the general property of detachable and attached types: an entity needs to be attached only if we perform a call on it. Since no calls are made on y and z, there is no need to declare them as attached (and to lock their processors).

3.3 Support for inheritance and polymorphism

Our technique is compatible with inheritance and polymorphism. Since T is a subtype of ?T, we may redefine a feature in a descendant class following Rule 1.

Rule 1. Result and argument redefinition.

- The return type of a feature may be redefined from ?T to T.
- The type of a formal argument may be redefined from T to ?T.

If the original version of the feature takes an argument of type **separate** *T*, we can redefine it in a descendant so that it takes an argument of type ?**separate** *T*. A client that uses the original class will need to pass an attached actual argument. Even if the redefined version of the feature is called (due to dynamic binding), that actual argument will conform to the required type. Obviously, we cannot redefine a detachable formal argument into an attached one – the type safety would not be preserved in the presence of polymorphism and dynamic binding. Note that the contravariant redefinition rule for the "detachability" of formal arguments (as opposed to the covariant rule for their class types) implies that a redefined version of a feature may lock at most as many arguments as the original one. In other words, the clients will not be cheated on – they may expect at most as much locking as specified by the signature of the feature; no additional locking may be introduced when redefining the feature.

There are, however, two problems related to the use of contravariant redefinition:

- The use of **Precursor** calls is not always possible.
- Inherited precondition and postcondition clauses that involve calls on redefined formal arguments may become invalid.

Consider the common programming pattern depicted in Figure 3. The redefined version of feature *r* lists precondition *new_precondition* that weakens the requirements put on clients (assume that the original feature is depicted in Figure 1). The body of *r* follows a simple pattern: if *new_precondition* holds, some particular actions corresponding to that new case are taken; otherwise, **Precursor** (*x*, *y*, *z*) is called. But this call will be rejected by the compiler because the types of actual arguments *y* and *z* (?**separate** *Y* and ?**separate** *Z*, respectively) do not conform to the types of the corresponding formals (**separate** *Y* and **separate** *Z*, respectively). In order to use calls to **Precursor**, explicit downcasts (object tests) must be performed.

```
r (x: separate X; y: ?separate Y; z: ?separate Z)
    require else
        new_precondition
    do
        if new_precondtion then
            -- do something here
        else
            Precursor (x, y, z) -- Invalid!
        end
    end
```

Fig. 3. Use of **Precursor**

While the problem of invalid precursor calls is easy to detect (it amounts to a simple type-check performed by the compiler) and to deal with, the second problem mentioned above – contract inheritance – is much trickier. Consider again the programming pattern used in Figure 3. The **else** part implicitly assumes that *some_precondition* holds because we know that *some_precondition* **or else** *new_precondition* holds and *new_precondition* is false. This assumption is valid if *some_precondition* does not involve calls on y or z. What happens if such calls do appear in *some_precondition*? For example, take *some_precondition* to be $x.$*is_empty* **and** $y.$*is_empty*. What is the meaning of $y.$*is_empty* in the context where y is of a detachable type? According to the call validity rule, call $y.$*is_empty* is valid only if the type of y is attached, which obviously is not the case here. Nevertheless, in the context of the inherited routine where y was attached, it was a valid call. So, it seems that we have a problem with contract inheritance – due to contravariant redefinition of formal arguments from attached to detachable, it is possible to invalidate inherited assertions that involve calls on redefined arguments. There are two simple solutions to this problem:

1. Ignore all inherited assertions that involve calls on detachable formal arguments, i.e. assume that these assertions hold vacuously. For example, $x.$*is_empty* **and** $y.$*is_empty* would reduce to $x.$*is_empty* **and true** hence to $x.$*is_empty* if y is detachable.
2. Prohibit the redefinition of formal arguments involved as targets of feature calls in preconditions and postconditions.

The first solution is compatible with the rules of Design by Contract when applied to preconditions – inherited preconditions are simply weakened. Unfortunately, postconditions may get weakened too, which is clearly against the rules of DbC. The second solution does not suffer from that drawback. Nevertheless, it forces the programmer to preserve the attached type of a formal argument even if the redefined version of the routine does not rely on any properties of that argument anymore. It might have no importance in the sequential context but in a concurrent context, where the detachability of an argument implies less locking, such restriction is very unwelcome. Essentially, once a formal argument has been used in a precondition or a postcondition, it cannot be redefined from attached to detachable in descendants. This means that there is no possibility to reduce the locking requirements of the routine.

In practice, we may expect that an attached separate formal argument involved in a postcondition will never be redefined into a detachable one, simply because all redefined versions of a routine have to satisfy the original postcondition (possibly strengthened) and there is no way to satisfy the postcondition without the guarantee that no other clients may change the state of the object represented by the formal argument. Such guarantee may only be obtained by locking the argument for the duration of the call which will only happen if the type of the argument is attached. On the other hand, it is logical that a redefined version of a routine that does not need to lock a given formal argument does not make any assumptions about the state of the object represented by that argument, i.e. it simply ignores the precondition clauses concerning that

argument. Therefore, we could combine both solutions presented above into one solution that is both sound (i.e. it follows the principles of Design by Contract) and flexible. We disallow the redefinition of a formal argument from attached to detachable if the inherited postcondition involves calls on that formal argument. No such restrictions are put on arguments involved in preconditions; if an inherited precondition clause involves a call on a detachable formal argument, that clause is considered to hold vacuously. We refine the rule for result and argument redefinition accordingly.

Rule 1'. Result and argument redefinition (refined).

- The return type of a feature may be redefined from $?T$ to T.
- The type of formal argument x may be redefined from T to $?T$, provided that no calls on x appear in the inherited postcondition.

3.4 Discussion

In addition to the solution based on attached types, we considered two alternative ways of specifying which formal arguments should be locked. The first solution is a compiler optimisation: if the body of r does not perform any calls on x, then the processor handling x does not need to be locked. The programmer does not need to use any additional type annotations to mark the arguments to be locked. Unfortunately, this solution is not acceptable for two main reasons:

- The client cannot see whether the formal argument is locked or not without looking at the implementation of the feature; the interface is not precise enough to infer all the necessary information.
- In the presence of polymorphism and dynamic binding the client might be cheated on – a redefined version of the feature might lock an argument that the original version does not lock.

The second solution relies on the extensive use of preconditions. In order to make sure that the processor handling x is locked throughout the execution of r's body, we need to include the assertion *is_available* (x) in the precondition clause. The fact that x is a formal argument of the routine does not automatically imply locking.

```
r (x: separate X; y: separate Y; z: separate Z)
    require
        is_available (x)
        ...
    do
        ...
    end
```

Such assertions are like wait-conditions (see 2.3) – they force clients to wait until the processor that handles the corresponding formal argument is available (i.e. it can be locked). This solution is compatible with polymorphism and dynamic

binding. Removing *is_available* (*x*) from the precondition clause of a redefined version of *r* eliminates the lock requirement on *x*'s processor. Such redefinition can be viewed as a particular case of precondition weakening which is a standard technique of Design by Contract. Although theoretically sound, this solution is not likely to be accepted in practice because it is too verbose and it puts too much burden on the programmer. Also, it is based on the special semantics for the assertion *is_available* which might be a bit misleading – programmers might think that *is_available* is a feature applicable to **Current**. Finally, as a matter of taste, it seems much easier to write (and read) code like this

```
s (x, y, z: separate X; a: ?separate A)
    do ...
    end
```

using the technique based on attached types, than clumsy code like that

```
s (x, y, z: separate X; a: separate A)
    require
        is_available (x)
        is_available (y)
        is_available (z)
    do ...
    end
```

The solution based on attached types is the only one that is theoretically sound, practical, and elegant. It also integrates best with other object-oriented mechanisms. We decided to propose it as the standard approach.

4 Lock passing

The next step to refine the access control policy and increase the expressiveness of the model is to allow clients to temporarily pass on their locks to their separate suppliers when needed. This was impossible to implement in the original SCOOP model where clients would keep exclusive locks during the execution of the routine that acquired the locks. Our approach relies on the mechanism described in section 3 – clients and suppliers use detachable and attached types to specify whether lock passing should take place. The proposed mechanism makes concurrent programs less deadlock-prone and allows programmers to implement interesting synchronisation scenarios.

4.1 The need for lock passing

In SCOOP, clients executing a routine that locks separate suppliers hold exclusive locks on these suppliers during the whole duration of the routine call. As pointed out in section 2.3, this policy ensures that no other client can jump in and modify the state of the supplier object between two consecutive calls issued

by our client. While such a guarantee is very convenient for reasoning about concurrent software – we may apply similar techniques as for sequential programs – it unnecessarily limits the expressiveness of SCOOP and leads to deadlocks. To illustrate the problems caused by the restrictive locking policy, we use a simple example in Figure 4. Calls to $x.f$, $x.g$, and $y.f$ are asynchronous (f and g are

```
r (x: separate X; y: separate Y)
    do
        x.f
        x.g (y)     -- x waits for y to become available.
        y.f
        ...
        z := x.some_query   -- Current waits for x.
                            -- DEADLOCK!
    end
```

Fig. 4. Deadlock caused by cross-client locking

commands), so the client will not wait for their completion. In fact, following the *wait-by-necessity* principle (see section 2.3), the client will only wait for the result of the query call $x.some\_query$. Unfortunately, this will cause a deadlock because the processor that handles x will not be able to evaluate *some_query* before finishing all the previously requested calls on x; it will not be able to execute $x.g\ (y)$ until it acquires a lock on the processor handling y but that processor is still locked by the client and it can only be unlocked once the client finished the execution of r's body. So, the client is waiting for x's processor and vice-versa; none of them will ever make any progress.

In fact, getting into a deadlock situation is even simpler. The client may simply pass itself as an actual argument to a separate query call, as in Figure 5. Since feature g called on x needs to lock the processor that handles **Current**, it will block until that processor is unlocked. But it will never be unlocked because it is waiting for the completion of the call to g. Again, we have a deadlock. This time, it is caused by a callback (or rather a "lock-back") of g's processor on **Current**'s processor. Note that the body of g does not even need to involve any real callback on **Current** in order to cause a deadlock.

```
s (x: separate X)
    do
        z := x.g (Current) -- x waits for Current; Current waits for x.
                           -- DEADLOCK!
    end
```

Fig. 5. Deadlock caused by a callback

Meyer [1] suggested that the problem depicted in Figure 5 could be solved by the use of the *business card principle* – clients may only pass the reference to **Current** to features that do not lock the corresponding formal argument, i.e. whose body does not contain any calls on that argument. Unfortunately, the business card principle does not work well with inheritance and polymorphism – it suffers from the same drawbacks as the first alternative approach to locking that we discussed in section 3.4. Also, it only solves the problem if there are no callbacks in the body of routine g. In the presence of actual callbacks, we would still end up with a deadlock.

Note that, in both examples, the deadlock occurs at the moment when the client waits for one of its suppliers. Since the client is waiting, it does not perform any operations on its suppliers. Therefore, it makes no use of the locks it holds. If the client could temporarily pass on the lock on y (in Figure 4) respectively on **Current** (in Figure 5) to its supplier x, the supplier would be able to execute the requested feature and return the result, which would allow the client to continue. We would be able to avoid deadlock. We use that observation to develop a lock passing mechanism that allows clients to agree to "lend" their locks to suppliers for the duration of a single separate call. The solution proposed by Brooke et al. [2] takes the opposite approach – it allows suppliers to get locks from clients without their consent. See section 5 for a comparison of both approaches.

4.2 The mechanism

We cannot simply say that locks are passed whenever possible as this would limit the number of synchronisation scenarios that can be implemented. In particular, some synchronisation scenarios supported by the original model would not be implementable in the extended SCOOP. Obviously, we want to preserve the backward-compatibility with the original model while making it more flexible and expressive. We want to give the programmers the possibility to decide whether lock passing should take place in a given situation or not. Once again, detachable types offer a simple solution. We introduce the lock passing rule based on the new semantics for detachable types and argument passing.

Rule 2. Lock passing. Assume that client c and suppliers x and y are handled by processors P_1, P_2, and P_3, respectively. If P_1 holds a lock on P_2 and P_3, and c makes a separate call $x.f\ (y)$ then, if the formal argument of routine f that corresponds to y is of an attached type, the call will be executed synchronously, with P_1 passing on all its locks to P_2 and waiting until the execution of f terminates, then revoking all its locks from P_2 and continuing its own execution.

Let us re-consider our examples. Feature r in Figure 6 is identical with feature r from Figure 4 but the semantics of argument passing follows Rule 2. As a result, call $x.g\ (y)$ will be executed synchronously, with the client passing on all its locks to x. No deadlock occurs at the moment when the client evaluates $x.some\_query$ because x is not blocked anymore, as it was the case in Figure 4.

```
r (x: separate X; y: separate Y)
    do
        x.f
        x.g (y)          -- Current passes its locks to x
                         -- and waits until g terminates.
        y.f
        ...
        z := x.some_query    -- No deadlock here!
    end
```

Fig. 6. Cross-client locking without deadlock

Similarly, the problem of separate callbacks can be solved thanks to lock passing. Routine s in Figure 7 is not deadlock-prone anymore because separate call $x.g$ (**Current**) results in lock passing that allows x to obtain a lock on **Current** without waiting. In this particular case Rule 2 has been applied taking $P_1 = P_3$ – the client and the actual argument are both **Current**, therefore they are handled by the same processor; we assume that every processor, when non-idle, implicitly holds a lock on itself. Note that, whenever lock passing occurs

```
s (x: separate X)
    do
        z := x.g (Current) -- x gets lock on Current from Current.
                           -- No deadlock here!
    end
```

Fig. 7. Callback without deadlock

as a result of a feature call, the client passes *all* its locks to the supplier, not only the locks on processors that handle the objects corresponding to the actual arguments of the call. This is because the client does not use any locks anyway while waiting until the execution of the supplier's feature has terminated. On the other hand, the supplier might require these additional locks in order to terminate the execution of the feature. Therefore, all locks are passed "just in case". Such generous behaviour of clients avoids more potential deadlocks than passing just the specified locks.

4.3 Lock passing in practice

We said that programmers should be able to decide whether lock passing takes place or not, and that the new locking policy should be backwards-compatible with the original SCOOP approach. This means that all scenarios supported by SCOOP should be easily implementable in the extended model.

This flexibility can be achieved through different combinations of detachable and attached types of formal and actual arguments of suppliers' features. Rule 2 states that lock passing only takes place if the corresponding formal argument is attached. From section 3.2 we know that an attached formal argument is locked by the routine. So, a routine that declares an attached formal argument will always lock that argument, either by waiting for the corresponding processor to become free (if its client does not hold the lock), or by enforcing lock passing from the client (if the client already holds the lock). Figure 8 illustrates these two cases. Since feature f in class X takes an attached argument, $x.f\ (y)$ will result in lock passing whereas $x.f\ (z)$ will be executed asynchronously without lock passing (just like in original SCOOP). This is because the client holds a lock on y but no lock on z.

```
-- in class C
z: separate Y
...
r (x: separate X; y: separate Y)
    do
        x.f (y)      -- Lock passing occurs because
                     -- Current has lock on y.

        x.f (z)      -- No lock passing because
                     -- Current has no lock on z.

        x.g (y)      -- No lock passing because
                     -- g takes detachable argument.

        x.g (z)      -- No lock passing because
                     -- g takes detachable argument.
    end

-- in class X
f (y: separate Y)
    do
        ...
    end

g (y: ?separate Y)
    do
        ...
    end
```

Fig. 8. Lock passing

If the called feature takes a detachable formal argument, as feature g in class X in Figure 8, no lock passing is performed. This logically follows from

the fact that such a feature does not lock the formal argument, as expressed by the locking rule in section 3.2. Since no lock is necessary, no lock passing takes place, independently of whether the client holds a lock on the corresponding actual argument (e.g. *y*) or not (e.g. *z*). Naturally, if a routine takes a detachable formal argument, it is possible to pass a detachable entity as actual argument (obviously, no lock passing takes place there because the client cannot hold a lock on a detachable entity). The opposite situation, i.e. passing a detachable actual argument to a routine that takes an attached formal argument, violates the conformance rules of detachable and attached – it is rejected by the compiler. Figure 9 recapitulates possible type combinations of formal and actual arguments and the resulting semantics of argument passing (*yes* stands for "lock passing takes place", *no* stands for "no lock passing").

	actual locked by client	actual not locked
actual attached, formal attached	yes	no
actual attached, formal detachable	no	no
actual detachable, formal detachable	no	no

Fig. 9. Lock passing combinations

Coming back to the problem of backward-compatibility of our approach with the original SCOOP model, we can see that our new semantics corresponds to the original one in case where actual argument is not locked by the client. On the other hand, if the client holds a lock on actual argument, our semantics differs from SCOOP's. Nevertheless, it is possible to emulate the original semantics – at a cost of some additional code – through the use of detachable formal argument and an auxiliary feature that takes an attached formal argument, as illustrated in Figure 10. The call to $x.f\ (y)$ does not block, even though the client holds a lock on *y*. The call to *blocking_f* will later block the supplier but it does not influence the execution of our client's code. Therefore, we obtain the semantics of the original SCOOP model. Note the use of object test for a downcast from detachable to attached in feature *f*.

5 Related work

This paper builds on our previous work on locking policy for SCOOP described in a technical report [8]. The report discussed the use of detachable types in SCOOP but did not cover the problems related to inheritance and polymorphism, such as contract inheritance. The lock passing mechanism was only described shortly, without considering more complex scenarios that we discuss in this paper. The report also presented a basic mechanism for shared locking based on a refined notion of pure query and a new semantics for **only** clauses. The shared-locking mechanism proved unsound in the presence of polymorphism, therefore we do

```
-- in class C
z: separate Y
...
r (x: separate X; y: separate Y)
    do
        x.f (y)      -- No lock passing because
                     -- f takes detachable argument.

        x.f (z)      -- No lock passing because
                     -- f takes detachable argument.
    end

-- in class X
f (y: ?separate Y)
    local
        y': separate Y
    do
        if {y': separate Y} y then
            blocking_f (y')
        end
    end

blocking_f (y: separate Y)
    do
        ...
    end
```

Fig. 10. Emulating original SCOOP semantics

not consider it in this paper. We are currently working on a refinement of that mechanism that provides full support for inheritance and polymorphism.

Meyer [7] discusses detachable types, in particular their use for eliminating *catcalls*. He also describes the idea of using detachable types in the context of SCOOP which was a result of our earlier discussions. He does not discuss the problem of feature redefinition in a concurrent context. Nevertheless, his solution of the catcall problem prompted us to dig into the issue of contract redefinition that is also relevant to SCOOP. To prevent catcalls, Meyer's rule for argument redefinition requires that if the class type of a formal argument is redefined covariantly, it must become detachable. Nevertheless, no restrictions are put on formal arguments that appear as call targets in inherited preconditions and postconditions; inherited assertions involving calls on detachable targets are evaluated using an implicit object test. For example, for attached x and detachable y, expression $x.is\_empty$ **and** $y.is\_empty$ is understood as $x.is\_empty$ **and** $(\{y'\!:\ Y\}y$ **implies** $y'.is\_empty)$, hence $x.is\_empty$ if y is void and $x.is\_empty$ **and** $y.is\_empty$ otherwise. Besides being complicated, this solution is inconsistent with Design by Contract – as we demonstrated in section 3.3, it

may lead to postcondition weakening. Our refined rule for result and argument redefinition (Rule 1') may be combined with Meyer's solution to ensure consistency with DbC and to simplify his approach. This shows that our technique, initially developed to solve concurrency issues, proves very useful in a sequential context as well.

```
-- in class C
r (x: separate X; y: separate Y)
        -- We assume that y = x.my_y, so both Current and x will
        -- call the same separate object.
    do
        x.f          -- Body of f will request lock on y.

        -- Will lock passing happen here?
        y.f
        -- Or here?
        y.g
        -- Or here?
        y.h
        -- Or here?
    end

-- in class X
my_y: separate Y

f -- Perform some calls on my_y.
    do
        ...
        s (my_y)     -- Snatch lock from client.
        ...
    end

s (y: separate Y) do ... end
```

Fig. 11. Problems with transitive locking

Brooke et al. [2] propose a CSP semantics for SCOOP. The authors identify the problem of repetitive locking and propose to solve it by applying transitive locking by default. That is to say, if client object c holds locks on supplier objects x and y and x requests a lock on y, x will temporarily "snatch" that lock from the client object. An advantage of transitive locking is that it offers more potential parallelism than our solution (we apply synchronous semantics to calls that involve lock passing). Nevertheless, the programmer has no control over lock passing; transitive locking is always applied, even if there is no danger of deadlocking. Furthermore, it is possible that calls on y issued by c and x are interleaved: even though c temporarily loses its lock on y, it is impossible to

predict when it happens. If c executes several calls on y after the call on x (see Figure 11), lock passing may occur either before the call to $y.f$, before the call to $y.g$, before the call to $y.h$, or after the latter. In fact, lock passing may even not happen at all – if the execution of routine f by x is very slow then the client object may be able to schedule all its calls on y and terminate the body of r before x tries to snatch the lock. As a result, one cannot assume the order of execution of separate calls; hence, assertional reasoning about separate calls is not possible. This problem is particularly acute in the context of inheritance and polymorphism: even if the original version of routine f in class X does not perform any calls that would lead to lock passing, a redefined version may do so. Clients of the original class are completely unaware of that and do not expect any lock passing. Our solution avoids such problems: clients make explicit decisions about lock passing; a redefined version of a routine cannot require more locks than the original version. Furthermore, Brooke et al. only allow locks on separate objects to be passed to the supplier. As a result, the separate callback depicted in Figure 5 still leads to a deadlock. Another difference w.r.t. our solution is the fact that only one lock is passed at a time; nevertheless, additional locks can be demanded by subsequent calls. The CSP model proposed by Brooke et al. may be extended to account for the differences mentioned above. In particular, every call involving lock passing should be treated similarly to a query call, i.e. it should be executed synchronously, with the client's handler being blocked until the call returns.

Rodriguez et al. [9] enrich JML with annotations for specifying atomicity and synchronisation constraints. Features can specify a list of locks that they acquire and release during their execution. A *locks* clause may appear in the specification of a feature after a precondition. By default, the locks clause has value *\nothing* for a non-synchronised feature. For features declared as synchronised, *locks* evaluates to *this* (or the class object if the considered feature is static). Another predicate, *lock_protected* (<*o*>), is added to the specification language. It allows to state that a given object (*o*) is protected by a (non-empty) set of locks, and that all these locks are held by current thread. Further, predicate *thread_local* (<*o*>) marks objects as thread-local, i.e. only reachable by current thread. Thread-local objects correspond to non-separate objects in SCOOP. Accesses to thread-local objects do not interfere with the activity of other threads, so they do not need to be synchronised. Although locking is specified at the feature level, it is more fine-grained than in our approach – locks are applied to single objects rather than whole processors. Also, lock passing is naturally supported. Nevertheless, the support for inheritance and polymorphism is lacking, e.g. it is possible to cheat on clients by performing more locking in a redefined version of a feature.

In Boyapati and Rinard's Parametrized Race-Free Java (PRFJ) [10], to gain an exclusive access to an object, a thread has to acquire the lock on the root of the ownership tree that contains the object. Every object has an owner: an object (possibly the object itself) or *thisThread*. If an object is owned by *thisThread* (directly or indirectly), it is local to the corresponding thread and it cannot

be accessed by other threads. Ownership is fixed, i.e. objects cannot change their owners over time. This ownership relation is very similar to the ownership relation between processors and objects in SCOOP, although the ownership structure in SCOOP is much simpler because objects cannot be owned by other objects and ownership is not transitive. A method may require callers to hold one or more locks before calling it – the locking requirements can be specified using the *requires* clause. Although PRFJ does not support Design by Contract, we may view *requires* annotations as part of routine contract. The support for inheritance and polymorphism is very limited – PRFJ does not offer the same flexibility w.r.t. routine redefinition as our approach. Also, unlike our approach, PRFJ does not support the combination of condition synchronisation and atomic locking of several objects.

6 Conclusions

We presented two simple refinements of the access control policy for SCOOP. We proposed a mechanism for specifying which arguments of a routine call should be locked. This mechanism, based on the novel concept of detachable types, allows for a precise specification of locking requirements, thus eliminating unnecessary locking that is often exhibited in SCOOP programs. We also introduced a lock-passing mechanism that allows clients to temporarily pass on their locks to separate suppliers. Both proposed mechanisms greatly improve the flexibility of the model and reduce the danger of deadlocks. They allow programmers to efficiently implement synchronisation scenarios that were difficult (or impossible) to implement in the original SCOOP model.

The *scoop2scoopli* preprocessor and the *SCOOPLI* library [1] support the lock-passing mechanism. We tested these tools in two iterations of a graduate course at ETH Zurich. A deadlock-detection scheme that supports lock-passing has also been devised and implemented [11] as part of SCOOPLI. We are currently working on the implementation of detachable types.

Our future research will be focused on the enhanced type system for SCOOP [12][13] and its applications to deadlock prevention. We think that the assertion language of Eiffel needs to be enhanced to allow for more expressive contracts for concurrency. In particular, we would like to enrich the specification of frame properties and use a refined notion of pure query to allow for safe interleaving of pure queries requested by different clients. We have already proposed the basic mechanism for shared locking [8] but more research is necessary to ensure its compatibility with the principles of Design by Contract.

7 Acknowledgements

Bertrand Meyer largely contributed to the development of the detachable/attached type mechanism and suggested its possible application in the context of

[1] Available for download at *http://se.ethz.ch/research/scoop.html*

SCOOP. Bernd Schoeller pointed out the problem of precursor calls in the presence of contravariance. We are grateful to the participants of the 2^{nd} SCOOP workshop for the discussions concerning the locking policy of SCOOP.

References

1. Meyer, B.: Object-Oriented Software Construction. 2nd edn. Prentice Hall (1997)
2. Brooke, P.J., Paige, R.F., Jacob, J.L.: A CSP model of Eiffel's SCOOP. Submitted for publication (2005)
3. Meyer, B.: Applying "Design by Contract". IEEE Computer **25** (1992) 40–51
4. Nienaltowski, P., Meyer, B.: Contracts for concurrency. In: First International Symposium on Concurrency, Real-Time, and Distribution in Eiffel-like Languages (CORDIE), York, United Kingdom (2006)
5. Caromel, D.: Towards a method of object-oriented concurrent programming. Communications of the ACM **36** (1993) 90–102
6. ECMA: Eiffel analysis, design, and programming language. ECMA Standard 367 (2005)
7. Meyer, B.: Attached types and their application to three open problems of object-oriented programming. In: European Conference on Object-Oriented Programming. (2005) 1–32
8. Nienaltowski, P.: Refined access control policy for SCOOP. Technical Report tr511, Computer Science Department, ETH Zurich (2006)
9. 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). (2005) 551–576
10. Boyapati, C., Rinard, M.: A parametrized type system for race-free java programs. In: Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). (2001)
11. Moser, D.: Design and implementation of a run-time mechanism for deadlock detection in SCOOP. ETH semester project (2005) available at http://se.inf.ethz.ch/projects/daniel_moser.
12. Nienaltowski, P.: Efficient data race and deadlock prevention in concurrent object-oriented programs. In: OOPSLA'04 Companion. (2004) 56–57
13. 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. Springer Verlag (2006)

Automatic Realizations of Statically Safe Intra-Object Synchronization Schemes 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

Abstract. This article presents the approach taken in MP-Eiffel to handle intra-object synchronization of concurrent objects. The correctness of concurrent objects is discussed assuming the existence of contract language mechanisms. Several synchronization schemes are presented and their automatic realization by the compiling system is discussed. A new proposal for automatic realizations of mixed synchronization schemes, either in exclusion or in concurrency, is presented and discussed. It is shown that one of those mixed schemes provides a solution for the safe integration of intra-object and inter-objects synchronization schemes within a concurrent object. The problem of conditional synchronization is also considered.

1 Introduction

MP-Eiffel[1] [1] is a prototype language which is being designed and implemented to test and validate concurrent object-oriented language mechanisms. Its goal is to provide a coherent group of expressive and safe concurrent language constructs suitable for general concurrent programming based on static typed systems, contracts, and pure[2] object-oriented languages.

Instead of developing from scratch a new object-oriented language, we have decided to base our work upon an appropriate existing language. The choice was (obviously) Eiffel [2] due to its simplicity, elegance, static type system, and because it is one of the (very) few languages with Design-by-Contract^{TM} (DbC) mechanisms [2, Chap. 9]. DbC mechanisms are very important not only to assert the correctness of (sequential) objects (and all their uses) [3, Chap. 11], but also to describe their semantics as instances of possibly partial Abstract Data Type (ADT) implementations [3, Chap. 6]. In concurrent programs it would be desirable to ensure that the same properties also apply to concurrent objects[3].

[1] Multi-Processor Eiffel.

[2] An object-oriented language is considered pure if its programs are only composed of communicating objects.

[3] A precise definition of concurrent objects will be presented in Sect. 2.4.

The first goal of this article will be to provide a definition of concurrent object correctness, taking into consideration not only its possible concurrent uses (which is a well established theory) but also the DbC mechanisms. The relation between (isolated) concurrent objects and class contracts is also established. Of particular interest will be the relation of concurrent contracts with the correctness and behavior of the (concurrent) object.

The second goal of this article is to assess the realizability of (internal) object synchronization schemes by the compiling system. We describe several synchronization schemes, identify the information that the compiling system must gather in order to allow their automatic realization, and present some algorithms used to implement them. The ability to automatically synchronize concurrent objects using different schemes enables the approach, recently adopted in MP-Eiffel[4], of detaching concurrent entity program annotation from specific synchronization schemes (unlike other object-oriented concurrent languages). As a result, MP-Eiffel objects might be tuned, after program development, to many different synchronization schemes in order to improve some of its properties such as liveness or the availability of concurrent objects.

A relevant contribution is the study done on the automatic realization of mixed synchronization schemes. In particular, one such scheme is originally proposed to solve the problem of integrating two different synchronization types in a concurrent object. One resulting from internal object safety concerns (intra-object synchronization), and the other from external client needs (inter-object synchronization).

2 Basic Definitions

For a better comprehension of this article, some basic terms and definitions will be presented here.

2.1 Service, Attribute, Command and Query

An object is composed by a set of **services**, which may be **attributes** (object data fields), or **routines** (object methods). When necessary, services might also be classified as **queries** and **commands**. Queries are services which return object properties (they are used to observe the object). Commands are services which may modify the object's state. Queries that never modify the object's state are called **pure**. Usually they are functions, although they can also be attributes. Commands are procedures (routines without return values, or *void* functions in languages that use C terminology).

In the context of object attachment, we will use the term **entity** to refer to identifiers in the class text that might. at run-time, become attached to objects. In Eiffel there are four types of entities [2, page 275]: attributes, local entities of routines, formal routine arguments, and the reference to the current object (**Current** in Eiffel and **this** in Java [4] and C++ [5]).

[4] In a previous version [1], concurrent objects always used a readers-writer exclusion mechanism.

2.2 Processor

It is common to use the term *thread* to represent program execution units which operate on a shared memory environment in the same computer. Likewise, *process* is frequently used for more decoupled execution units (in UNIX a process can share its address space with multiple threads). Since those terms are usually connected with specific operating system execution units, and because we are interested in abstract concurrent programming in which each execution entity may be implemented in many different ways (even through different processes in several computers on a distributed network), we will use the abstract notion of **processor** adapted from Meyer [3, page 964]:

> A processor is an autonomous thread of control capable of supporting the sequential execution of instructions.

A processor might be implemented as a process, a thread, a group of processes in a network of distributed computers, or in any other realizable way.

In the context of the execution of services in objects, a processor will be called **writer** if it is executing a service that might change the object's state (usually commands); and it will be called **reader** if it is executing side-effect free services (pure queries).

2.3 Models of Inter-Processor Communication

There are two basic models for inter-processor[5] communication: message passing (direct)[6]; and shared memory (indirect).

In message passing inter-processor communication, by definition, the sender (caller) processor will always be different from the receiver (callee) processor. The ACTOR family of languages [6] and SCOOP[7] [7, 3] restrict their concurrent communication mechanisms to this model (ADA95 [8] also has a message passing mechanism named rendezvous). In the original proposal of SCOOP each concurrent object is handled by a single processor throughout its entire life (which makes its synchronization very simple).

In shared memory communication the caller processor is the same as the callee processor. Concurrent programming systems that use this communication model are the POSIX-THREADS [9] library, ADA95 protected types, and the language JAVA [4].

MP-EIFFEL has language constructs for both types of communication models. Message passing is expressed by triggers and remote entities, and shared memory through shared and remote entities (a detailed description of these mechanisms can be found in [1])[8].

---

[5]

[6] Not to be confused with object-oriented inter-object message passing communication. Both share the message passing communication model semantics, but applied to different communicating parties.

[7] Simple Concurrent Object-Oriented Programming.

[8] Section 2.4 briefly defines shared and remote entities.

2.4 Concurrent Objects

In a concurrent object-oriented program it is essential to identify all the objects – named concurrent – that (might) require a proper synchronization scheme in order to be correctly and safely used. Therefore, a concurrent object will be 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. All objects that are not concurrent are named sequential.

In the case of inter-processor message passing communication mechanisms all the objects able to directly handle requests from different processors are concurrent. In SCOOP, the static typed system is used to conservatively identify concurrent objects. A concurrent annotation (**separate**) is used to identify all the entities to which concurrent objects might be attached. Its properties [3, pages 973–975] ensure that concurrent objects cannot be unsafely attached to sequential entities and sequential objects to concurrent entities.

The identification of concurrent objects when using shared memory inter-processor communication mechanisms is, in general, much more difficult. One possibility (which could also be applied to the other communication model) is to delegate such responsibility on the programmer, as happens, for example, in JAVA. We are not interested in such error prone unsafe approach to concurrency. Instead we want to statically identify (even if conservatively) all possible concurrent objects, and, using such knowledge, to automatically synchronize those objects ensuring their correctness and safety. Like SCOOP, MP-EIFFEL uses the static typed system to unambiguously identify all possible concurrent objects. To that goal two new type annotations were added to the EIFFEL type system: **shared** and **remote** (entities with one of these annotations are called concurrent entities). Shared objects are concurrent objects that can be observed and modified by different processors. Remote objects are also concurrent objects with the restriction that there is only one "writer" processor (a unique processor is allowed to modify its state). The language type rules [1] ensure the safe use of concurrent and sequential objects.

2.5 Intra-Object and Inter-Object Synchronization

A concurrent object might be required to meet different synchronization needs. On one hand, an object is required to protect itself from concurrent executions of its services. This type of synchronization is named **intra-object synchronization**.

On the other hand, clients might require the exclusive use of concurrent objects throughout the execution of more than one of their services. This synchronization type, which results from external uses of the object, is named **inter-object synchronization**.

In concurrent objects in which both synchronization types are required, it is necessary to ensure that at least one general solution (automatically implementable) exists, which allows their correct integration regardless of the intra-object synchronization scheme used. Such solution exists, and will be presented in Sect. 4.5.

2.6 Conditional Synchronization

A service from a concurrent object may not always be available to be used by clients. Often, services are usable only if certain conditions on the object's observable state are met. For example, a request for a `pop` service on a concurrent `STACK` only make sense if the stack is not empty. A possible solution to this problem is to use a **conditional synchronization** scheme in which the client is required to wait until the condition is met (wait condition).

2.7 Concurrent Assertions

An assertion is said to be concurrent if it contains at least one concurrent assertion clause[9]. 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.

2.8 Concurrent Object Availability

In order to compare different intra-object synchronization schemes, it is useful to have some kind of objective metric expressing the ability for an object to be executed concurrently. That is the purpose of the Concurrent Object Availability metric.

Considering that N_x is the maximum number of processors sharing some property x operating in an object OBJ (for example, reading or writing properties), and that N_c is the maximum number of such processors which can safely operate concurrently inside OBJ (of course: $N_c <= N_x$), we define the Concurrent Object Availability of OBJ in relation to the processors with the x property as being:

$$COA_x = \frac{N_c}{N_x} \tag{1}$$

This factor measures the maximum percentage of processors with some property that can safely operate concurrently inside an object.

It should be mentioned that this factor may not be unique in each synchronization scheme, and may depend on the concurrent state of the object (for example, the use of an object by processors with a certain property may exclude its usage by other type of processors).

3 Concurrent Object Correctness

Object-Oriented software construction is defined by Meyer [3, page 147] as the building of software systems as structured collections of possibly partial abstract

[9] An assertion clause is a simple boolean condition declared inside an assertion.

data type (ADT) implementations. Therefore, the correctness of an object-oriented program depends mainly on the correctness of each of the ADT's it implements, regardless of the possible complex interactions they might occur between them. A necessary condition for a concurrent object to be correct is that its ADT is never compromised by its concurrent use.

Viewing objects as instances of ADT implementations simplifies their use, and provides a solid theoretical basis for object-oriented programming. It should be noted that objects – being instances of ADT implementations – should include the ADT semantic properties. Those properties can be expressed by class invariants, and service preconditions and postconditions. Unfortunately few languages allow the implementation of these semantically rich ADTs, by including mechanisms to express, and when possible, to test those assertions. EIFFEL is one such rare language (it pioneered this ADT view of objects, promoting the Design by Contract programming methodology), but several other tools are beginning to appear, for example, in JAVA [10] and C++ [11].

In sequential programs, in order not to compromise the correctness and simplicity of ADTs, objects can be externally used only at their stable times [3, page 364]. Such a behavior is relatively easy to ensure in sequential programming, because there is only one processor. However, in concurrent programs in which there is the possibility of intra-object concurrency the problem is much more complex. In the presence of invariants, stable times can only be enforced within the class boundaries if it is forbidden the existence of public modifiable attributes (otherwise, any client could change the object's state, possibly breaking the invariant, outside of its control).

3.1 Linearizability

A sufficient condition to ensure the correctness of concurrent objects is linearizability [12, 13].

Linearizability

> An object is linearizable if each operation appears to take effect instantaneously at some point between the operation invocation and response.

3.2 Class Contracts

In contract aware languages, linearizability is required to take also into consideration possible executable class assertions which are also applicable to concurrent objects.

A sufficient condition to ensure the correctness of concurrent objects with class assertions, is to consider the (possible) execution of all the applicable assertions as being part of the linearizable object operation.

3.3 Concurrent Contracts

The existence of concurrent objects raise another very interesting problem: besides normal sequential assertions, class contracts can instead make use of concurrent assertions.

In such cases, what should be its correct behavior?

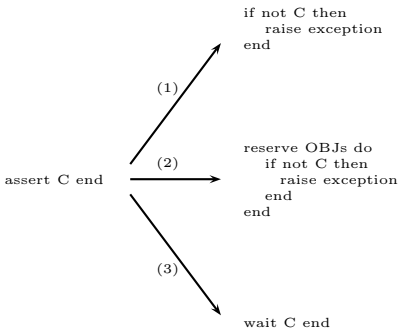


Fig. 1. Possible behaviors of concurrent assertions

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 Fig. 1) could clearly create race conditions, hence it is an unacceptable behavior.

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

The verification of any assertion is always of the responsibility of the program code that is (or could be) executed before the assertion. Hence, preconditions are the responsibility of object clients, and invariants and postconditions on the object itself. In a sequential program there is only one processor, so if an assertion is proved (usually by testing it) to be false, then we are clearly in the presence of a programming error, because its value will remain false unless, afterwards, the processor itself does appropriate actions to change it. However, concurrent programs have more than one processor. So if the assertions is concurrent then its state can change without the participation of the processor which is doing

its runtime verification. So again, the only safe behavior will be to ensure that concurrent assertions behave as wait conditions.

In the case of intra-object synchronization, the only concurrent assertions that are relevant are preconditions and (eventually) invariants (when tested before service execution).

Concurrent contracts (and also concurrent conditions within structured conditional and iterative instructions) are also of key importance for inter-object synchronization, but such discussion is beyond the scope of this article (a draft article on inter-object synchronization in MP-Eiffel can be found in [14]). Also beyond the scope of this article is the possible dynamic nature of concurrent assertions. The same assertion, depending on the context of its verification, might behave concurrently or sequentially (see [14]).

3.4 Total Object Covering

A trivial necessary condition regarding intra-object synchronization is the requirement that the object's synchronization scheme covers all of its external services. In the absence of this restriction, most likely race conditions would arise on the access of the object's attributes, compromising, in an unpredictable way, the correctness of the object's ADT implementation.

Total Object Covering

> A correctness condition for the synchronization of concurrent objects is the necessity that all of the object's exported services are protected with an appropriate synchronization mechanism.

One of the strongest objections of Brinch Hansen [15] to the concurrent mechanisms of the language Java is the fact that its programs may not observe this rule, posing safety problems.

4 Intra-Object Synchronization Schemes

Having defined the essential correctness conditions and requirements to observe in the synchronization of concurrent objects, in this section we will present several synchronization schemes with sufficient realizability conditions to allow an automatic safe implementation by the compiling system.

4.1 Monitors

A simple and sufficient approach to ensure linearizability, is to consider each object to be a monitor (Fig. 2). Interestingly, Hoare [16] and Brinch Hansen [17] themselves have recognized the importance of the class concept of the first object-oriented language, Simula, when they proposed the monitor synchronization scheme.

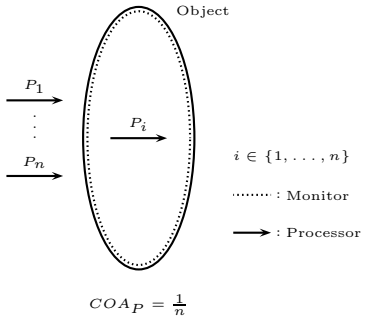


Fig. 2. Monitors

Monitors are the simplest synchronization scheme. The price to pay for that simplicity is that objects objects synchronized with a monitor are only available to one processor at a time. For n processors the monitor COA value is $\frac{1}{n}$, which is its lowest possible useful value.

The concurrency mechanisms of JAVA were initially designed to be approximations of monitors [18, page 399], but their intents have failed in some important aspects [15]. The current version of the language [4], although not solving some of the original monitor problems, allows, although to a limited extent, the use of other synchronizing schemes besides monitors [19].

Realizability Monitors pose relatively few requirements on the compiling system. A trivial requirement is the necessity for identifying all of the object's public services. Those services will need to be protected with monitor synchronization code.

A sufficient algorithm to implement this synchronization behavior, is to create a proxy class with an identical interface of the unsynchronized class, in which the monitor synchronization code is implemented. This approach has also the advantage of avoiding the over-synchronization problem of calling public services inside the object.

Monitors implement conditional synchronization through the use of condition variables. A possible (though inefficient) algorithm would be to attach a single condition variable to the monitor (object) and to signal all waiting processors (broadcast) at the end of the object's public routines. It would also be necessary to convert each concurrent precondition (and invariant) to a wait instruction and relevant code on the condition variable.

As an example, appendix A.2 presents a possible automatic implementation of a monitor synchronization scheme of a stack class (described in A.1). As is easy to verify, the automatic translation of the stack class (into `MONITOR_STACK`) requires little semantic knowledge on the part of the compiling system. Although the presented translation algorithm takes advantage of the ability to distinguish commands and impure queries from pure queries; it is not a monitor requirement but simply an optimization of the conditional synchronization mechanism.

Section 5 discusses other more efficient possibilities to optimize conditional synchronization algorithms.

4.2 Readers-Writer Exclusion

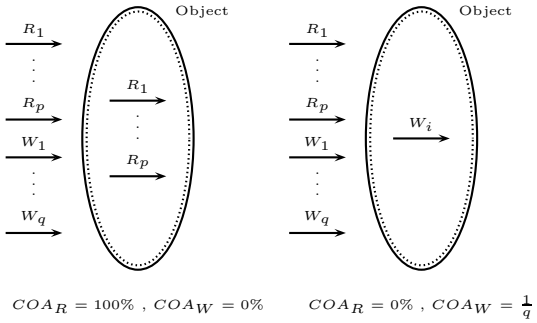


Fig. 3. Readers-Writer Exclusion

The imposition of mutual exclusion for processors requiring the execution of services in concurrent objects, may be considered an overwhelming restriction. Frequently, some of the processors are only trying to query (without side-effects) the object to get some information. In these cases, it is sufficient to ensure mutual exclusion only when a service with side-effects on the object state (usually commands) is being processed, allowing the concurrent processing of the remaining (pure query) services.

Hence an approach using the synchronization scheme of readers-writer exclusion [20] (a writer processor excludes all the others, but multiple reader processors can operate concurrently) is also a valid and safe choice (Fig. 3). This scheme has higher average COA values than monitors, hence is less prone to block the access of concurrent objects, which may also reduce the risk of some global (program wide) liveness problems such as deadlocks.

This synchronization scheme is used in the language Ada95 (protected types), and was also the first approach taken by the prototype language MP-Eiffel, proposed by the author.

Realizability This synchronization scheme is "better" (higher average COA values) than monitors, but the compiling system requires a little more information on the concurrent object class. Unlike monitors, this scheme requires the ability to distinguish commands and impure queries from pure queries.

In MP-Eiffel this is implemented through the following reasoning. A service is considered as having side-effects if its program includes an assignment instruction to one of the object's attributes, or if there is a call to a routine with side-effects. In qualified calls to routines we take the conservative approach of

verifying the purity of all possible dynamic binding routines. Recursive routines (either directly or through other routines), pose no problem to this approach because the compiling system keeps track of the routines already traversed.

Conditional synchronization differs from the monitor case due to the fact that there are two different lock instructions (one for reading and one for writing). Other than that, the algorithm can be quite similar.

Appendix A.3 presents a readers-writer exclusion possible automatic implementation of the stack class.

4.3 Concurrent Readers-Writer

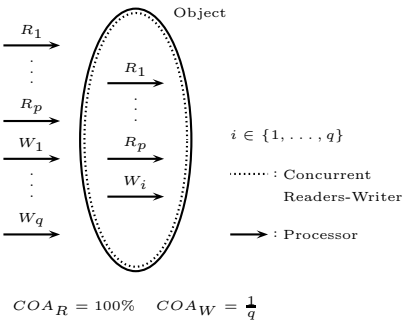


Fig. 4. Concurrent Readers-Writer

Lamport [21] has proposed a generalization to the previous synchronization scheme which allows the concurrent access of multiple reader processors and a single writer processor. Mutual exclusion is only required to multiple writer processors (Fig. 4). In this way, reader processors never block a possible writer processor.

In Lamport's proposal, in order to ensure that a reader processor is done in object stable times (when the invariant holds), the requested service (query) is repeated whenever it occurs concurrently with a writer operation.

In the integration of this scheme within objects it is necessary to foresee the situation of an invariant failure in the beginning of the execution of one, or more, "reader" services resulting simply due to a concurrent execution of a "writer" service. This possibility needs to be properly taken care, imposing, for example, the repetition of the "reader" services when the invariant fails and a concurrent "writer" access has been detected (otherwise, the invariant should indeed fail).

This scheme is very interesting due to the fact that, in its implementation, it does not impose much more restrictions than those imposed by the previous scheme. It has less contention (higher or equal COA value) in the execution of writer services which reduces the risk of deadlocks. However, it may create starvation problems in the reader services when the execution of writer services is overwhelmingly frequent [21, 22].

A possible solution, in some cases, to this problem is proposed by Peterson [22]. The main idea is based in the duplication of the object's state.

In the important particular case in which there is only a unique processor with the possibility to execute writer services[10], Peterson [22] proposes a wait free algorithm to any processor which makes the object always available ($COA = 100\%$) to any processor.

Realizability Lamport's synchronization scheme maintains the requirements imposed by the reader-write exclusion scheme, extending them with the necessity of allowing possible repetitions of reader services.

This repetition (hidden from the object's clients) does not pose serious implementation and semantic problems because – by definition – reader services don't change the object's state. However, it is necessary to foresee the situation in which there might exist assertion failures (invariant, preconditions or others) during the reader execution, resulting from changes on the object state produced by a concurrent writer execution. Hence, this synchronization scheme requires a language in which it is possible to transparently catch all the exceptions created during the execution of services, allowing to verify if the failure cause was due to the interference of a concurrent writer service – in which case it can be ignored and the execution of the reader service has to be repeated – or if it is a real correctness failure. This restriction is essential to the implementation of this scheme, because it is the only way to distinguish real failures from harmless (in this particular case) race-condition ones.

Conditional synchronization can be similar to the monitor's case.

Appendix A.4 presents a proxy class with an implementation of this algorithm.

4.4 Lock-Free Synchronization

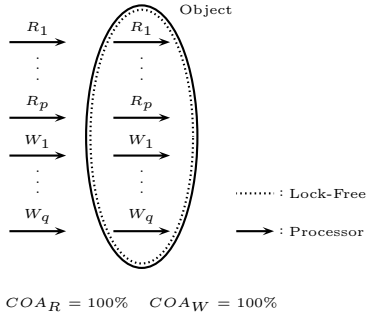


Fig. 5. Lock-Free Synchronization

[10] This situation occurs in MP-Eiffel when remote entities are used in shared memory communication mechanisms.

A group of synchronization schemes that has been deserving a growing enthusiastic interest are lock-free and wait-free synchronization [23] (Fig. 5). This type of synchronism is characterized by the assurance that processors are able to execute operations on shared resources, regardless of the execution time of other processors, always with the guarantee that at least one of them will always be successful (an important particular case is wait-free synchronization, in which it is ensured that all processors will be able to perform the requested operation in finite time).

The advantages of this approach are the inexistence of processor blocking[11] (making it immune to deadlocks) and tolerance to faults on other processors. These characteristics make it especially suitable for real-time programming [24].

Currently, this type of synchronism is seldom used, though this situation is expected to change in the future. A good sign towards that direction was the public release of a library of classes for JAVA (JSR 166: Concurrency Utilities [25]) that use this type of synchronism.

The reasons why lock-free synchronization is so rarely used are its complexity, the specificity and low level of many of its algorithms, and also because it is difficult to ensure safe implementations. Here we are interested in a preliminary study on the the possibility for future automatic safe implementations of lock-free synchronization schemes.

Basic Concepts In general, lock-free synchronization algorithms are based on the total, or partial, duplication of the object's attributes and, when necessary, in concentrating all of the necessary modifications to that object in a unique atomic modification. Usually this atomic object state modification is implemented using special hardware instructions, such as the instructions `CAS` (*Compare-And-Swap*) or `LL/SC` (*Load-Linked, Store-Conditional*). In those algorithms it is necessary to foresee and accept possible failures (due to the action of another concurrent processor). When this happens, it is necessary to repeat the entire process. In the special case of wait-free algorithms, as mentioned before, a limit to the maximum number of repetitions is ensured.

Herlihy [26, 23] has demonstrated that there are universal algorithms able to implement this synchronism in concurrent objects observing the linearizability condition, presenting also universal methodologies (though not very efficient) [26, 27] for its implementation. The presented methodology, as mentioned by Herlihy, is adaptable to be automatically executed by the compiling system.

Other possible lock-free related algorithms are based on software transactional memory [28]. Those algorithms work in a analogous way as transactions in database systems. Transactions proceed in three steps. First a transaction is announced, then the executions of the required operations is performed, and finally an attempt to commit the transaction is performed. On failure, it is ensured that the the transaction did not change the memory state. Otherwise, the results of the transaction take (atomic) effect. This transaction process is repeated until it succeeds. Harris and Fraser [29] proposed a language mechanism

[11] Except when conditional synchronization is required.

for JAVA (strongly based on Hoare's conditional critical regions) which takes advantage on the possibilities of software transactional memory for general lock-free algorithms (it also includes a mechanism for conditional synchronization). If the requirements imposed on the compiling system presented ahead are met, Harris and Fraser implementation can be safely used to implement a lock-free synchronization scheme in concurrent objects (in order to do that, it is required that the atomic construct is applied to the whole public services of the object).

Realizability Either Herlihy's generic algorithm [27] or the software transactional memory algorithm, require the ability to take copies of the state of objects, and the necessity of allowing possible repetitions of services. This last requirement, is the one which restricts the most the static safe realizability of these algorithms.

In fact, even taking into consideration that the execution of a service by a processor apply to a separate stable copy of the object local to that processor, not all services can be repeatedly executed without nasty side-effects to other processors (and the system's state). For example, a service which invokes a writing routine to an external terminal device (or for that matter: to any external file), or which reads information from external users, cannot be safely repeated.

On the other hand, services which only modify the values of attributes are repeatable.

Service repetition

> A service is repeatable if its effect in the system's state – program or eventual external entities depending on it – as a result of its execution, is discardable as if the service did not execute.

Hence, this synchronization scheme is only statically realizable in a safe way if the compiling system is able to correctly identify all the repeatable services of each concurrent object.

It should be noted that, unlike the previously presented synchronization schemes, lock-free algorithms are not yet integrated and conveniently experimented in MP-EIFFEL (hopefully that situation will change in the near future).

Table 1 summarizes the most important requirements posed on the compiling system by the presented synchronization schemes.

Table 1. Compiling system requirements posed by non-mixed schemes

	Monitors	Readers-Writer Exclusion	Concurrent Readers-Writer	Lock-Free
Concurrent object identification	**Yes**	**Yes**	**Yes**	**Yes**
Pure query identification	**No**	**Yes**	**Yes**	**Yes**
Repeatable pure queries identification	**No**	**No**	**Yes**	**Yes**
Repeatable services identification	**No**	**No**	**No**	**Yes**

4.5 Mixed Synchronization Schemes

So far we have been looking into concurrent object's synchronization as if it required a unique uniform synchronization scheme. However, there is no theoretical reason for not considering the possibility of using different synchronization schemes, simultaneously or alternating in time, within a concurrent object; in an attempt, for example, to increase its concurrent availability. As it will be seen, this approach will also provide a safe generic solution for reserving concurrent objects (which is an inter-object synchronization problem) for exclusive uses of its services, without compromising its intra-object synchronization scheme (which could even be lock-free).

Naturally, such mixed combinations of synchronism have to observe all the required correctness conditions, including – in particular – the necessity of total object covering.

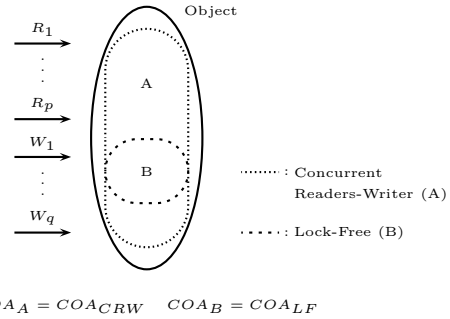


Fig. 6. Example of a mixed synchronization scheme

Mixed Exclusion Schemes One possible way of combining several synchronization schemes is to impose mutual exclusion between them. For example, an object can possess a group of services which could, within themselves, be synchronized by a lock-free scheme, and others that, due to not being repeatable, require mutual exclusion, readers-writer exclusion or concurrent readers-writer (Fig. 6) schemes with all of the object's services. In this situation it would be perfectly safe to use an asynchronous group mutual exclusion mechanism [30]. Using this mechanism, several processors can concurrently access the lock-free services, but in mutual exclusion with the remaining processors attempting to execute other services.

Another situation with a similar solution occurs when we are interested in having different synchronization schemes depending on the context in which the object is externally used. For example, in MP-Eiffel an object can be reserved to be used exclusively by a single processor for a sequence of calls to its services [1] (inter-object synchronization). If that object happens to have, for instance,

a lock-free synchronization scheme, then both uses would not be possible, limiting the usability of more powerful synchronization schemes. A solution to this problem is to use a mixed scheme with both synchronizations, implemented with a group mutual exclusion mechanism to prevent the simultaneous use of both schemes. In this way, we are able to safely switch, at run time, between different synchronization schemes in the same object, making full use of less restricting schemes.

Correctness of Mixed Exclusion Schemes

It is safe to use any combination of mixed exclusion schemes if the following conditions are observed:

a) Total object covering;
b) Each of the synchronization schemes are safe within the the part of the object it applies (which is a subset of all the object's services).

The demonstration of this correctness condition is straightforward. Since the mechanism of group mutual exclusion, by definition, ensures that at most only one of the synchronization groups is active, and being also ensured that all of the object's services are synchronized by at least one group (there could be more than one), it is easy to conclude that it is sufficient to make sure that each group of synchronization schemes is safe in the subset of services to which it applies.

Mixed Concurrent Schemes By definition, the vast majority of mixed concurrent combinations schemes are not safe. A concurrent modification of concurrent object attributes leads almost always to race conditions in their access from which can result, in an unpredictable way, senseless incorrect values for those attributes, breaking the class's invariant.

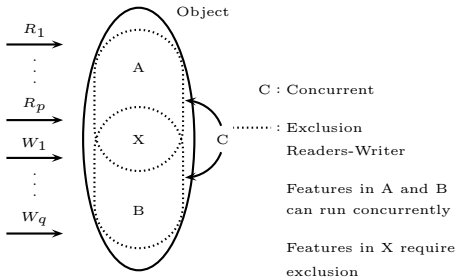


Fig. 7. Double readers-writer exclusion

However, in certain particular cases it looks like it could make sense to allow, in a very disciplined way, the concurrent access to the object, even without

requiring lock-free or readers-writer concurrent synchronization schemes. For example, the use of two or more concurrent groups of mutual or readers-writer exclusion, (Fig. 7) within an object – each one protecting the access to a separate group of attributes – not being in general safe since nothing ensures that in such a situation the invariant will hold when tested, can be linearizable if some restrictions are imposed.

Using a real life example analogy, if we have a `CAR` object it would be safe to concurrently replace a tire and change its oil, without necessarily having to impose a lock-free scheme (that is, without the necessity of imposing the repeatability of either of those operations).

Since in this article we take the semantically rich view of objects as being instance of ADT implementations with executable assertions. The implementation of these schemes are required to safely verify all of the object's class assertions. Lets take a closer look to the correctness requirement of an object's service `S` [3, pages 368–370]:

$$\{INV \text{ and } PRE_S\}\ BODY_S\ \{INV \text{ and } POST_S\}$$

So the execution of an object service will be correct if, before its execution, the class invariant and the service precondition are true, and, afterwards, the same happens to the invariant and the service postcondition.

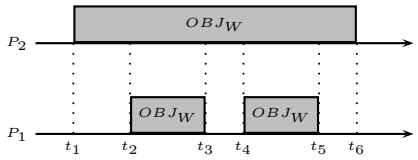


Fig. 8. Wrong execution in an object with mixed concurrent synchronization

Assuming, for the sake of the argument, only calls to writer services (OBJ_W), the execution presented in Fig. 8 is not linearizable, since the processor P_1 cannot safely test the class's invariant in the interval $[t_3, t_4]$ between its calls to object's services.

Linearizable invariant verification

Taking a closer look at the Fig. 8, several considerations can be drawn. From the point of view of processor P_1 it would be linearizable to anticipate the invariant verification from the instant t_2 to the instant t_1, since if the invariant holds in t_1 it would also hold in t_2 if there wasn't the interference of processor P_2. So, it would be perfectly acceptable to reuse the invariant test done by P_2 in t_1, to the processor P_1 (meaning to assume the invariant of t_1 in t_2).

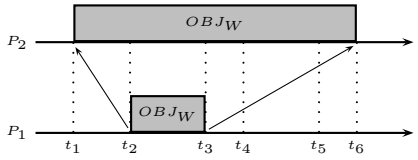


Fig. 9. Correct execution in an object with mixed concurrent synchronization

In a similar way, it would be linearizable to delay and reuse the invariant test done by P_1 in t_3 to P_2 in t_6, if, meanwhile, no more calls to object services on behalf of P_1 are allowed (Fig. 9)[12].

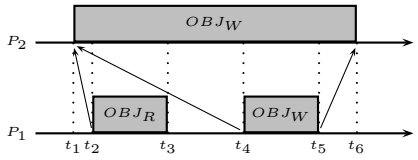


Fig. 10. Correct execution in an object with mixed concurrent synchronization

On the other hand, the situation presented in Fig. 10, although it involves two service executions by processor P_1 concurrently with one execution of P_2, can be considered safe, since the invariant cannot change during reader (OBJ_R) service calls, which is why, the invariant verified in the instant t_1 can be consistently reused in instants t_2, t_3 and t_4.

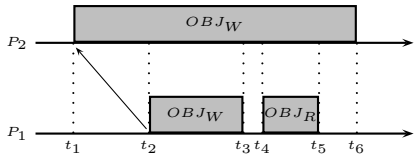


Fig. 11. Wrong execution in an object with mixed concurrent synchronization

[12] This behavior affects exception handling, but this problem dealt in MP-Eiffel goes beyond the scope of this article.

The case presented in Fig. 11 is not correct since when the processor P_1 begins a reader service execution in t_4, it is not possible to reuse nor to verify the class's invariant.

To complete the analysis to this type of synchronization schemes, there are two situations that need to be taken care of. The first one occurs when the first concurrent execution is done by a reader service. In this case, it is easy to conclude that the invariant, from the point of view of the processor executing that service, will be the same at the end of the execution. Hence, the execution of this type of services is irrelevant to the correctness of the mixed concurrent synchronization schemes, and so, can be "ignored".

Finally, the first writer service entering a concurrent execution zone, need not to be the last writer to leave (as happens in the figures shown). What needs to be imposed is that the "input" invariant to be reused, will be the one in the beginning of the execution of the first writer, and that the "output" invariant to be the one occurring at the end of the last writer.

Generalizing all those cases:

Concurrent Verification of Invariants

In a concurrent execution of several processors within an object in the presence of mixed concurrent synchronization schemes, it is linearizable to verify the invariant only when the first writer processor begins, and the last writer processor finishes, if in that interval, the following conditions hold:

a) Any processor may execute all the reader services it wants, as long as all of them precede a possible invocation of a writer service by the same processor;
b) Each processor may only invoke a single writer service;
c) After the execution of a writer service, a processor may not execute any other service.

Getting back to the `CAR` example, with a mixed concurrent scheme with multiple readers-writer exclusion groups respecting the above conditions, it would be possible to concurrently replace a tire and change its oil by different employees (processors), but restricting each employee to only perform one operation concurrently with the other employees. That is, an employee can only proceed its work on the car if it is ensured that the last one was done correctly, thus not compromising the car's invariant. It is not hard to conclude that these considerations are generalized to the concurrent mixture of other types of synchronization schemes.

Correctness of Mixed Concurrent Schemes

It is safe to concurrently mix two or more synchronization schemes as long as the following conditions are observed:

a) Total object covering;
b) Each synchronization scheme protects a separate group of object's attributes;
c) The criterion for concurrent verification of invariants is observed.

Realizability One interesting characteristic of mixed schemes is the fact that the requirements posed by each scheme don't need to apply to the whole object, but only to a subset of its services.

The implementation of **mixed exclusion schemes** requires that each scheme is implemented only to the subset of services to which they apply (in some cases, it can be the whole object), and, as already mentioned, the use of the mechanism of asynchronous group mutual exclusion [30].

In the case of **mixed concurrent schemes**, the compiling system needs to gather more information about the program. In particular, it needs to know the subset of the object's attributes used, directly or indirectly, by each one of the services of concurrent objects. This information is essential in order for a static correctness verification of the application of this scheme. Only services that never interfere with each other may execute concurrently. All of the remaining services are required to execute in mutual or reader-writer exclusion with the services using the same attributes.

To implement a **mixed concurrent scheme** synchronization algorithm, it is sufficient the use of a simple approach based in a shared atomic counter. The appendix B has a possible safe implementation in a C alike language of this algorithm for the particular situation in which processors are POSIX-THREADS. In this implementation, all the necessary synchronization is done in the invariant verification, so reader and writer services only need to call the appropriate invariant testing functions. For writer services, each executing processor[13] not only reuses the invariant test done by the first processor, but also finishes its execution only when the last processor terminates the invariant verification. This ensures that at most, each processor only executes one writer service, and also that no reader execution is done afterwards. Reader services reuse the last tested invariant done after a writer service without blocking. It should be noted, that this implementation does not take into consideration conditional synchronization. However, since mixed concurrent schemes do not share attributes, the implementation of conditional synchronization if each mixed scheme, is guaranteed to work within the subset of services to which the scheme applies.

[13] Implemented only as POSIX-THREADS.

5 Conditional Synchronization Optimizations

In Sect. 4 and in appendix A we have presented a simple but inefficient automatic implementation of conditional synchronization. In this section we will discuss several possibilities to optimize such implementations.

A processor may wait until a synchronization condition is verified in two possible ways. Either by continuously testing the condition until it is verified (busy-wait); or it can go to sleep and rely on appropriate awaking signals when the activity of other processors might affect the expected condition.

The first technique might be an acceptable solution if a hardware central processing unit (CPU) is exclusively dedicated to the waiting processor (the CPU would be idle anyway); otherwise this behaviorless waiting process would be a meaningless waste of CPU cycles. We won't consider it here.

The second technique is much more economic use of the CPU resources, but it requires appropriate awaking signals external to that waiting processor.

When the responsibility for implementing correctly this type of synchronization is put on the programmers hands (such as in POSIX-THREADS), the programmer can decide when signaling is required to happen and, most importantly, to whom it should be addressed to. Monitors [16] were designed with this mechanism, in which condition variables were used to distinguish signals (POSIX-THREADS C library implements a similar concept).

In JAVA's approach, the responsibility for testing the synchronization conditions, and for waiting and signaling (notification) threads[14] is also delegated to the programmers hands. However, unlike the signaling mechanism of monitors and POSIX-THREADS, JAVA in its base mechanism lacks the possibility for fine-tuning the signaling sending process taking into consideration different synchronization conditions. There is a single waiting and signal mechanism which applies to the whole object (using POSIX-THREADS terminology, there is only one condition variable per object). Hence, a notify signal awakes a waiting thread regardless of its waiting condition (increasing the probability of spurious thread awaking). If there are multiple threads awaiting for different synchronization conditions, there is the possibility for a signal to awake the wrong processor, hence missing a correct destination. Due to this problem in JAVA, for safety concerns, it is advised to use broadcasts (signals to all waiting processors) instead of signaling a unique processor. This strategy leads to more possible spurious thread awaking, increasing the inefficiency of this mechanism.

In this article we are interested in implementations of automatic statically safe conditional synchronization mechanisms. Hoare [16] presents a possible automatic awakening mechanism (based on conditional waits), which he acknowledges as being simpler than condition variables, but with much less efficient implementations. That algorithm consists on sending broadcast signals after the execution of any of the external concurrent object's services, in order to impose a verification of all conditional waits in all waiting processors.

[14] Since we are now referring to JAVA, to avoid misunderstandings we will use its thread terminology instead of processor.

This algorithm can be made much more efficient, if the programming language has the ability to distinguish commands and queries. In that case, it is necessary to signal all waiting processors, only after the execution of commands (and eventually, also after non-pure queries), as they are the only ones who may change the object's state.

Currently this is the approach that we have been following, in the implementation of MP-Eiffel.

In this specific problem, the operational approach to concurrency – in which programmers are required to program directly the synchronization of concurrent objects – despite its unsafety, is still the one which allows much more efficient implementations of conditional synchronization awaking algorithms.

Nevertheless, two possible alternative approaches are being studied which may provide safe algorithms to this problem, which approximate the efficiency of hand made programmer algorithms.

One approach, extends the usefulness of concurrent assertions also for signaling purposes. If a concurrent assertion in a precondition is required to be a conditional waiting mechanism, then their occurrence in postconditions or in other internal assertions (such as checks), could be used for the complementary awaking process. In the example of concurrent stacks given above, a `pop` service is required to conditionally wait for a non-empty stack. However, the postcondition of the `push` service is precisely the non-emptiness of the stack, hence it seems a perfect fit. A possible algorithm to implement this approach, consists in assigning condition variables to each different concurrent assertion in preconditions. Then, the waiting process in those preconditions would simply be a wait operation on the respective condition variable. The signaling process, on the other hand, requires a little more work. First it is necessary to express all possible concurrent assertions (in preconditions, postconditions or in other assertions[15]) in relation to their individual concurrent assertion clauses. The signaling algorithm could then take advantage of that static knowledge, to signal only the conditional variables in which its precondition contains identical assertion clauses as those in the signaling postcondition or check assertion.

This approach, though promising, has the drawback of requiring appropriate use of concurrent assertions by the programmer, which may pose problems if a normal class (such as a stack) is to be reused for concurrent objects.

Another promising approach, requires a deeper introspection inside the object's code. The idea is to attach to each of the object's services, the static set of attributes which may be modified by the service execution. Likewise, a set of depending attributes is attached to each concurrent assertions of preconditions. The algorithm would then be to signal, at the end of each service, all the preconditions which may depend on the attributes possibly modified by the service.

[15] Except invariants, as they are required to be always observed in the object's stable times.

Acknowledgments

I would like to thank João Rodrigues and Tomás Oliveira e Silva for their invaluable help and suggestions. I would also like to thank all the reviewers for their comments and suggestions which have improved the clarity and contents of the article.

References

1. Oliveira e Silva, M.: Concurrent object-oriented programming: The MP-Eiffel approach. Journal of Object Technology: Special issue: TOOLS USA 2003 **3**(4) (2004) 97–124
2. Meyer, B.: Eiffel: The Language. Prentice Hall, Englewood Cliffs, N.J. (1992) 2nd printing.
3. Meyer, B.: Object-Oriented Software Construction. 2nd edn. Prentice Hall (1997)
4. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification. Third edn. Addison-Wesley (2005)
5. Stroustrup, B.: The C++ Programming Language. third edn. Addison Wesley Longman (1997)
6. Agha, G.A.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, Massachusetts (1986)
7. Meyer, B.: Systematic concurrent object-oriented programming. Communications of the ACM **36**(9) (1993) 56–80
8. U.S. Government: Ada 95 Reference Manual (Language and Standard Libraries). (1995)
9. Butenhof, D.R.: Programming with POSIX Threads. Addison-Wesley (1997)
10. Kramer, R.: iContract-the JavaTM design by ContractTM tool. In: Proceedings of Technology of Object-Oriented Languages – TOOLS 26. (1998) 295–307
11. Edwards, S., Sitaraman, M., Weide, B., Hollingsworth, E.: Contract-checking wrappers for C++ classes. IEEE Transactions on Software Engineering **vol.30**(11) (2004) 794–810
12. Herlihy, M.P., Wing, J.M.: Axioms for concurrent objects. In: POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM Press (1987) 13–26
13. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3) (1990) 463–492
14. Oliveira e Silva, M.: Concurrent contracts and inter-object synchronization in MP-Eiffel. Draft version available at `http://www.ieeta.pt/~mos/pubs` (2006)
15. Brinch Hansen, P.: Java's insecure parallelism. ACM SIGPLAN Notices **34**(4) (1999) 38–45
16. Hoare, C.A.R.: Monitors: an operating system structuring concept. Communications of the ACM **17**(10) (1974) 549–557
17. Brinch Hansen, P.: Monitors and concurrent pascal: a personal history. In: The second ACM SIGPLAN conference on History of programming languages, ACM Press (1993) 1–35
18. Gosling, J., Joy, B., Steele, G.: The Java Language Specification. First edn. Addison-Wesley (1996)
19. Lea, D.: Concurrent Programming in Java. Second edn. Addison-Wesley (2000)

20. Courtois, P.J., Heymans, F., Parnas, D.L.: Concurrent control with "readers" and "writers". Communications of the ACM **14**(10) (1971) 667–668
21. Lamport, L.: Concurrent reading and writing. Communications of the ACM **20**(11) (1977) 806–811
22. Peterson, G.L.: Concurrent reading while writing. ACM Trans. Program. Lang. Syst. **5**(1) (1983) 46–55
23. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems (TOPLAS) **13**(1) (1991) 124–149
24. Anderson, J.H., Jain, R., Ramamurthy, S.: Wait-free object-sharing schemes for real-time uniprocessors and multiprocessors. In: Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97). (1997) 111–122
25. Sun Microsystems, Java Specification Requests: JSR166: Concurrency Utilities (2004) (`http://www.jcp.org/en/jsr/detail?id=166`).
26. Herlihy, M.: A methodology for implementing highly concurrent data structures. In: PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming, ACM Press (1990) 197–206
27. Herlihy, M.: A methodology for implementing highly concurrent data objects. ACM Transactions on Programming Languages and Systems (TOPLAS) **15**(5) (1993) 745–770
28. Herlihy, M., Luchangco, V., Moir, M., William N. Scherer, I.: Software transactional memory for dynamic-sized data structures. In: PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing, ACM Press (2003) 92–101
29. Harris, T., Fraser, K.: Language support for lightweight transactions. In: OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications, ACM Press (2003) 388–402
30. Joung, Y.J.: Asynchronous group mutual exclusion. Distributed Computing **13**(4) (2000) 189–206
31. Oliveira e Silva, M.: Thread-Safe SmallEiffel (version beta-4) (2003) (`http://www.ieeta.pt/~mos/thread-safe-se/index.html`).

A Example Code

The code presented here is pure testable EIFFEL and was compiled with the thread-safe SmallEiffel package developed by the author (available in [31]).

A.1 Stack

```
-- Generic unbounded STACK class

deferred class STACK[E]

feature

  count: INTEGER is
      -- Number of elements
    deferred
    end;

  empty: BOOLEAN is
    do
      Result := count = 0
    end;

  top: E is
      -- STACK's last pushed element
    require
      not empty
    deferred
    ensure
      same_count:   count = old count
    end;
```

```
  push(elem: like top) is
    deferred
    ensure
      one_more:   count = old count + 1;
      element_placed_in_the_top:   top = elem;
    end;

  pop is
    require
      not empty
    deferred
    ensure
      one_less:   count = old count - 1
    end;

invariant

  count >= 0;
  empty = (count = 0)

end -- STACK
```

A.2 Stack: Monitor

```
class MONITOR_STACK[E]

creation
  make

feature {NONE}

  stack: STACK[E];
  mtx: MUTEX;
  cnd_var: CONDITION_VARIABLE;

feature

  make(s: STACK[E]) is
    require
      s /= Void
    do
      stack := s;
      create mtx.make;
      create cnd_var.make
    end;

feature

  count: INTEGER is
    do
      mtx.lock;
      Result := stack.count;
      mtx.unlock
    end;

  empty: BOOLEAN is
    do
      mtx.lock;
      Result := stack.empty;
      mtx.unlock
    end;

  top: E is
    do
      mtx.lock;
      from until not empty loop
        cnd_var.wait(mtx)
      end;
      Result := stack.top;
      mtx.unlock
    end;

  push(elem: like top) is
    do
      mtx.lock;
      stack.push(elem);
      mtx.unlock;
      cnd_var.broadcast
    end;

  pop is
    do
      mtx.lock;
      from until not empty loop
        cnd_var.wait(mtx)
      end;
      stack.pop;
      mtx.unlock;
      cnd_var.broadcast
    end;

end -- MONITOR_STACK
```

A.3 Stack: Readers-Writer Exclusion

```
class RW_EXCLUSION_STACK[E]

creation
  make

feature {NONE}

  stack: STACK[E];
  rwl: READ_WRITE_LOCK;
  mtx: MUTEX;
  cnd_var: CONDITION_VARIABLE;

feature
```

```
  make(s: STACK[E]) is
    require
      s /= Void
    do
      stack := s;
      create rwl.make;
      create mtx.make;
      create cnd_var.make
    end;

feature

  count: INTEGER is
    do
      rwl.read_lock;
      Result := stack.count;
      rwl.read_unlock
    end;

  empty: BOOLEAN is
    do
      rwl.read_lock;
      Result := stack.empty;
      rwl.read_unlock
    end;

  top: E is
    do
      rwl.read_lock;
      from until not empty loop
        rwl.read_unlock;
        mtx.lock;
        cnd_var.wait(mtx)
        mtx.unlock;
        rwl.read_lock;
      end;
      Result := stack.top;
      rwl.read_unlock
    end;

  push(elem: like top) is
    do
      rwl.write_lock;
      stack.push(elem);
      rwl.write_unlock;
      cnd_var.broadcast
    end;

  pop is
    do
      rwl.write_lock;
      from until not empty loop
        rwl.write_unlock;
        mtx.lock;
        cnd_var.wait(mtx)
        mtx.unlock;
        rwl.write_lock;
      end;
      stack.pop;
      rwl.write_unlock;
      cnd_var.broadcast
    end;

end -- RW_EXCLUSION_STACK
```

A.4 Stack: Readers-Writer Concurrent (Lamport)

```
class RW_CONCURRENT_LAMPORT_STACK[E]

creation
  make

feature {NONE}

  stack: STACK[E];
  mtx: MUTEX;
  writer_in,writer_out: INTEGER;
  cnd_var: CONDITION_VARIABLE;

feature

  make(s: STACK[E]) is
    require
      s /= Void
    do
      stack := s;
      create mtx.make;
      create cnd_var.make
    end;

feature

  count: INTEGER is
    local
      success: BOOLEAN;
      v: INTEGER
    do
      from until success loop
        v := writer_in;
        Result := stack.count;
        success := v = writer_out
      end;
    rescue
      if v /= writer_out then
        retry
      end
    end;

  empty: BOOLEAN is
    local
      success: BOOLEAN;
      v: INTEGER
    do
      from until success loop
        v := writer_in;
        Result := stack.empty;
        success := v = writer_out
      end;
    rescue
      if v /= writer_out then
        retry
      end
    end;
```

```
  top: E is
    local
      success: BOOLEAN;
      v: INTEGER
    do
      from until success loop
        v := writer_in;
        from until not empty loop
          mtx.lock;
          cnd_var.wait(mtx)
          mtx.unlock;
        end;
        Result := stack.top;
        success := v = writer_out;
      end;
    rescue
      if v /= writer_out then
        retry
      end
    end;

  push(elem: like top) is
    do
      mtx.lock;
      writer_in := writer_in + 1;
      stack.push(elem);
      writer_out := writer_out + 1;
      mtx.unlock;
      cnd_var.broadcast
    end;

  pop is
    do
      mtx.lock;
      from until not empty loop
        cnd_var.wait(mtx)
      end;
      writer_in := writer_in + 1;
      stack.pop;
      writer_out := writer_out + 1;
      mtx.unlock;
      cnd_var.broadcast
    end;

end -- RW_CONCURRENT_LAMPORT_STACK
```

B Mixed Concurrent Schemes

B.1 Invariant Testing Implementation

```
#include <pthread.h>

typedef struct
{
  int counter;
  int done_start;
  int Result_start;
  int Result_end;
  pthread_mutex_t mtx;
  pthread_cond_t cnd;
} INVARIANT_SYNCH;
#define INVARIANT_SYNCH_INIT \
  {0,0,0,0,PTHREAD_MUTEX_INITIALIZER,PTHREAD_COND_INITIALIZER}

int command_test_invariant(int (*inv)(void *obj),void *obj,
                           INVARIANT_SYNCH *synch,int start_of_routine)
{
  int Result;

  pthread_mutex_lock(&synch->mtx);
  if (start_of_routine)
  {
    synch->counter++;
    if (!synch->done_start)
    {
        // Invariant checked only in the first routine
        // (except for creation command, instead of rechecking
        //  the invariant, we could reuse the last Result_end).
      synch->Result_start = (*inv)(obj);
      synch->done_start = 1;
    }
      // Invariant result reused for all concurrent routines
    Result = synch->Result_start;
  }
  else // end_of_routine
  {
```

```
    synch->counter--;
    if (synch->counter == 0)
    {
        // Invariant checked only in the last routine
      synch->done_start = 0;
      synch->Result_end = (*inv)(obj);
        // awake all waiting processors (barrier end)
      pthread_cond_broadcast(&synch->cnd);
    }
    else
    {
        // wait for the last routine
      while(synch->counter > 0)
        pthread_cond_wait(&synch->cnd,&synch->mtx);
    }
    Result = synch->Result_end;
  }
  pthread_mutex_unlock(&synch->mtx);

  return Result;
}

int query_test_invariant(int (*inv)(void *obj),void *obj,
                         INVARIANT_SYNCH *synch)
{
  int Result;

  pthread_mutex_lock(&synch->mtx);
    // fetch last invariant verification
  if (synch->done_start)
    Result = synch->Result_start;
  else
    Result = synch->Result_end;
  pthread_mutex_unlock(&synch->mtx);

  return Result;
}
```

B.2 Reader Services Implementation

```
1.    if (!query_test_invariant(...))
1.1.    raise_invariant_exception(...);
2.    if (!test_precondition(...))
2.1.    raise_precondition_exception(...);
3.    Result = execute_query_body(...);
4.    if (!test_postcondition(...))
4.1.    raise_postcondition_exception(...);
5.    if (!query_test_invariant(...))
5.1.    raise_invariant_exception(...);
```

B.3 Writer Services Implementation

```
1.    if (!command_test_invariant(... ,1))
1.1.    raise_invariant_exception(...);
2.    if (!test_precondition(...))
2.1.    raise_precondition_exception(...);
3.    execute_command_body(...);
4.    if (!test_postcondition(...))
4.1.    raise_postcondition_exception(...);
5.    if (!command_test_invariant(... ,0))
5.1.    raise_invariant_exception(...);
```

Reliable Distributed Eiffel Components

Emmanuel Bouyer, Gordon Jones

Eiflex Ltd. - http://www.eiflex.com
emmanuel.bouyer@eiflex.com
gordon.jones@eiflex.com

Abstract. This paper addresses the issues of fault tolerance in distributed object systems, in particular in enterprise integration and process automation environments. It describes Eiflex – a framework for ensuring continuity of service and no loss of data integrity in the presence of failure of parts of a distributed object system. Eiflex offers a variety of interaction mechanisms between distributed objects: RPC – both synchronous and asynchronous; publish/subscribe; and reliable messaging. It also offers a lightweight persistence mechanism for transaction state. Eiflex is written in Eiffel and although doesn't use SCOOP, does use a similar apartment style of separation between its distributed component objects.

1 Introduction

1.1 Fault tolerance in distributed systems

All distributed systems have the potential for parts of a system to fail in isolation. A processor failure, a communications failure, a software failure, a planned maintenance outage. At some time disjoint parts of a distributed system will be unable to reach other parts. With simple client/server applications, particularly where the end user is a human at a GUI, if the server fails the human may well be rather frustrated but will usually adopt a strategy of shutting down, and trying again sometime later. However with multi tier or peer to peer computer linked systems, where each component has "in memory" state and is acting its part in a grand scheme, what should the isolated, but still alive, parts do in these partial failure circumstances? They could all respond to the failure by behaving like the simple client server case above, that is give up and hope they will be restarted later, or they could have inbuilt degradation and recovery mechanisms that manage the state and ensure continuity of service.

This paper presents the solution to these reliability issues adopted by the Eiflex framework, a distributed component toolset written in Eiffel, targeted at enterprise integration environments. In these partial failure situations, Eiflex offers a range of reliability options from graceful degradation to full hot standby failover.

The Eiflex framework has been used mostly as a software integrator. An enterprise bus, joining disparate application islands together using an "adapter" to talk each alien technology. Consequently, not only do Eiflex components have to contend with failures to reach other Eiflex components, they also have to contend with failing to

reach non Eiflex based applications which typically have been written without fault tolerance in mind.

1.2 Background of Eiflex

The resilience origins of Eiflex derive from a system commissioned by the Chicago Board of Trade (CBOT) futures and options exchange to support real time price dissemination from its trading floor to wallboards around the trading floor, and to traders desks in their offices. Trading applications are not allowed to stop running.

Fig. 1. The financial trading floor at CBOT with the wallboards in the background

The first version of the CBOT system went live in 1999, but the middleware (Eiflex) at its core has since evolved, as the moves to electronic based trading demanded far higher throughput, and as it was also used in other application domains.

A faster multi-threaded version of Eiflex went live at CBOT in 2004. Further performance improvements, and changes to give wider scope are still being undertaken. Most of this paper describes the target architecture of the distributed middleware, but there is also a short section describing why the various changes have been made since the first version.

1.3 Key attributes

Eiflex is a distributed object system. To meet the fault tolerance objectives, particularly when acting as an enterprise bus, or process integrator, the Eiflex distributed objects supply the following key attributes:

Transactional and Persistent components
Transactions and persistence are the traditional mechanisms for recovery from failure. However most schemes involve the use of a database for the preservation of long term persistent data, and the software façade in front of the database typically does not hold mission critical "in memory" data between transactions. The domains that Eiflex is aimed at are somewhat different. When it is acting as an application integrator, it will frequently hold medium term data to manage the mismatch of representation or processing sequence between the various applications it is integrating. The data Eiflex components are holding is mission critical, but it is relatively short lived, surviving just long enough to pipeline information from one application to others. A database is not really a suitable vehicle for capturing this medium term data, it is too heavyweight, and would impose potential bottlenecks in the integration activity.

Instead Eiflex components offer lightweight transactions and persistence, and has an implied assumption that data in transit has a target long term data store. Eiflex components are the sorting offices. They wont losing your packages, but they are not the final destination. There are 2 variants of the persistence: local and remote replication. Which is used depends on how fast the system must fully recover from failure.

Distribution recovery
If connection is lost between components, then no human interaction is required to re-establish connection. The "find and bind" mechanisms kick in automatically to re-connect either to the resurrected distributed objects (if only local persistence is in use), or to hot standbys (if remote replication has been exploited). The means by which processes are resurrected is described in section 4.3.

Adaptable
Given its prime role as an integrator, there is no limit to the different types of application island that Eiflex components might need to talk to, and so there is an adapter framework to straightforwardly integrate new "adapters" into a system, and for these to follow the same failover and recovery patterns as recovery for native Eiflex components. This adaptation aspect of Eiflex is beyond the scope of this paper, but suffice to say that the approach taken is to use deferred Eiffel classes for the adapter, and separate connection, reader and writer threads in the same way that Eiflex components interact with each other.

1.4 An example application

The following use of an earlier version of Eiflex highlights the key attributes above very clearly.

At the height of the dotcom bubble, Swedish company OM introduced in London UK a retail equities exchange called Jiway trading in international equities. The exchange guaranteed to match the best price available. If the exchange was unable to match buyers and sellers from its own books, it needed to send out hedge orders to other parities. The OM Jiway exchange was looking for technology to automatically

route these hedge orders from its central OM CLICK based exchange to partners providing order execution via the FIX protocol, and to return to the central exchange the results of these executions.

Superficially the functional aspects of this application are not particularly complex. Take some orders, break them down by price, farm them out to the appropriate execution providers, collect the results, and when all completed, return the results back to the central exchange.

However the non-functional aspects were severe. It must handle the mismatch of data between OM CLICK and FIX. It must never lose any of the intermediate data for partially executed orders, it must continue in the presence of broken connections, and continuously attempt to re-establish these connections using alternative routes and standbys. And of course it must always be available.

Jiway approached a number of suppliers both internally and externally. They eventually chose Eiflex as the solution because it was clear that its infrastructure provided the non-functional requirements as a given. At the time Eiflex did not have adapters to talk to either OM CLICK nor FIX, but it was judged that adding these to Eiflex was a far more cost effective approach than adding the resilience aspects to existing CLICK and FIX products.

The system had distributed component objects for:

- Wrapping each execution provider, each of which had a FIX adapter..
- Market-level and global configuration information. This was largely statically configured and published information about which providers traded in which equities.
- 'Hedge' engines, which performed the main processing for accepting the hedge orders using a CLICK adapter, breaking the order down by price, submitting them to the target execution provider with a FIX adapter, accepting the fills from the provider, consolidating the results, and feeding those back to central exchange via the CLICK adapter.
- Market Events component for controlling what state the market is in according to a daily schedule, and also controlling the firing of certain housekeeping activities.
- An analytical component providing statistics and audit trails.

Unfortunately, when the dotcom bubble burst, and retail trading of equities did not really take off, Jiway was closed, and so this excellent demonstration of the benefits of Eiflex is no more.

1.5 Asynchronous pipelining style

What is perhaps most relevant about the above example is the use of pipelining, intermediate state and asynchronous requests rather than distributed transactions. This style of application integration is where Eiflex is most relevant, and it is this style which is very appropriate to enterprise bus, and process automation domains. This model discourages the use of synchronous activity between distributed objects on the grounds that they can be both the causes of deadlock and bottlenecks.

2 The Channels library

The distributed component objects in Eiflex are known as channels with channel proxies referencing channel servers. The reason for the name channel is solely because one of the authors is French, one is English, and we coined the term channel tunnel as the interconnect (and Manche does not trip off the tongue so easily).

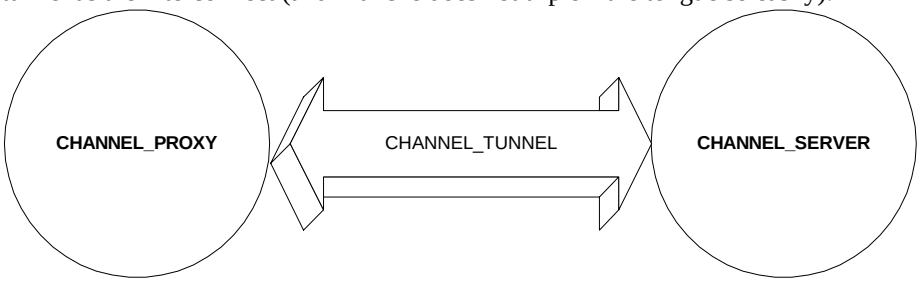


Fig. 2. Proxy and server interconnected via a tunnel. The use of depth in the arrow is meant to highlight that the infrastructure classes that implement the tunnels operate in a multithread environment, and use the thread cluster synchronization facilities offered by Eiffel Software. Whereas the channel proxy and server objects operate within a single thread – that in which they were created.

Channels are obliged to have unique (business oriented) identifiers to facilitate the find and bind of the proxy to the server. Unlike CORBA and SCOOP it is not (currently) possible to pass around distributed channel object addresses. Instead channel proxies are first class objects, and when a channel server or proxy is passed as an argument or in a topic, this results in a self binding channel proxy being constructed at the receiving end.

When a channel proxy is created it will immediately attempt to bind itself to the corresponding channel server object, and keep itself bound even if the channel server moves due to failure or redeployment (see the registry later). This dynamic bind mechanism is one of the main reasons there is no object address type, since the proxy may well find itself bound to a substitute after failure of a server.

When a channel server is created, it will immediately "register" itself (and keep itself registered) so that proxies can find it (see the registry later).

Eiflex is more like CORBA than SCOOP in its attitude to creating distributed components. A client cannot directly create a server object except via a factory object in the potentially remote process/thread. Every Eiflex server process is equipped with a channel thread factory and every channel thread is equipped with a factory for creating the domain specific channel server objects. Both of these factories are of course themselves channel servers.

The decision to deny SCOOP-like separate creation using a SCOOP-like CCF mechanism derives both from the fault tolerance requirements, and also so that distributed object location is under programmer control. Eiflex server objects have a life cycle that may include resurrection or being substituted by a clone after untimely death. Giving the client the responsibility for remotely creating the remote object and managing that life cycle does not seem desirable nor even feasible in the presence of broken connections.

2.1 Proxy server interaction

The dining philosopher classes at the end include examples of most of the concepts described here, so it is worth looking at the 2 classes as each concept is described.

Meta types

Eiflex transports Eiffel objects over proxy server interactions, however to allow for reflection, and to proved the application with intercept capability on (un)marshalling, it has its own meta-type mechanism, which is used to describe the structure of the object values being transported. This is not dissimilar in concept to IDL in CORBA and SOAP in the XML world. It differs in that it does not require a pre-compilation phase like CORBA (the meta type is built dynamically from an object or a class), and the in(ex)ternalization is more efficient than xml representation. Every channel server publishes 3 topics which describe (using this meta-type) all the operations (RPC), topics (pub/sub) and messages that the server offers.

Eiflex uses its own data marshalling driven by the meta-type system rather than the in(ex)ternalisation of objects supplied by the compilation system for a number of reasons[1]:

- The distributed components may not necessarily have the same classes inside them representing the exchanged business objects. Obviously the receiving type must have a conformant attribute hierarchy to that being sent, and so it certainly makes it much easier if both ends do use the same classes.
- The persistence and replication recovery has to be business neutral. So the information capture and recovery is very definitely value oriented, without any implied behaviour.
- Currently it is not feasible to extract reflection information for Eiffel routines, and Eiffel does not have native concepts for topics nor messaging.
- The scheme offers the ability for application specific intercepts (called storers) to marshal and un-marshal values of specifically registered classes and their descendents. In fact Eiflex uses this technique itself so that common data structures like hash table and list are treated in a library independent way, so that for example one executable can be using the Eiffel Software data structure library, but another can be using gobo. Eiflex also uses the same storer scheme to ensure any attempt to export a channel server object (or descendent) is changed to export the information to construct a proxy at the remote end.

The receiver of object values from channels must sometimes supply a meta-type to describe the data that is expected. Typically that metatype is created from a class name, but can also be produced from an object directly, and there are some common singleton types.

[1] We note that Eiffel Software are planning a revised int/externalisation library for a future release, and it is possible that some of the reasons we had to invent our own scheme may become redundant.

```
create {TTC_EIFFEL_TYPE}.make_from_classname (a_classname: STRING)
create {TTC_EIFFEL_TYPE}.make_from_object (an_object: ANY)
Eiffel_pathname_id_type
```

Type conformance between proxy and server depends on contracts and explicit tests rather than compile time protection. When un-marshalling, there is an explicit test that the created object conforms to the meta type expected by the receiving end.

There is a current restriction (expressed as a contract) that a marshaled object with no storer has no reference loops. Note however that a storer can be used to overcome this restriction on a case by case basis.

The interactions

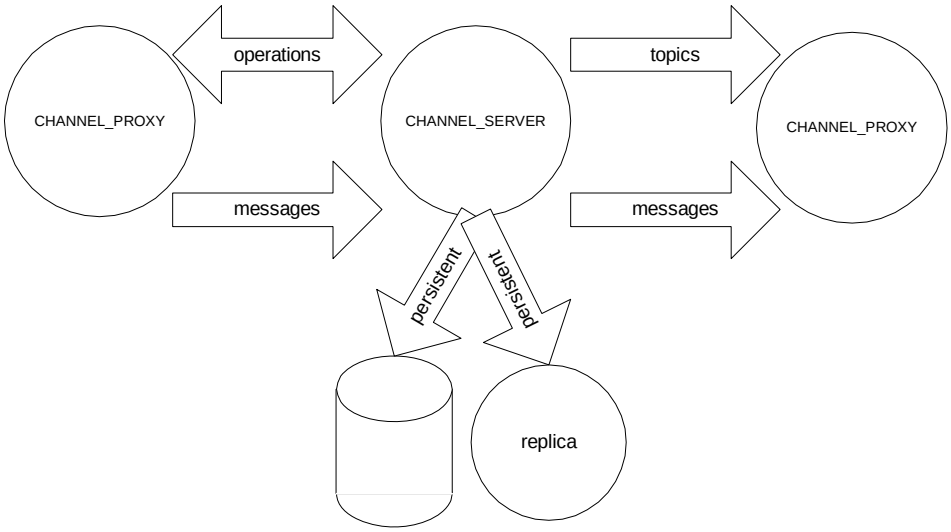


Fig. 2. Eiflex not only offers Remote Procedure Call (known as operations in Eiflex) as the means of communication between proxy and server, but in addition other forms of interaction are supplied, described in more detail below

Operations and requests

Channel servers supply operations for Remote Procedure Call.

```
Class TTC_OPERATION
…
make (
    a_name: PATHNAME_COMPONENT
    an_arg_type: TTC_EIFFEL_TYPE
    a_response_type: TTC_EIFFEL_TYPE
    an_action: FUNCTION [ANY, TUPLE [ANY], ANY] ) is…
```

Each time the operation is performed, the action function is passed an Eiffel object created by an_arg_type and populated by the supplied argument object. On return the response type is used to marshal the returned object using a_response_type.

Each call starts and ends a transaction.

On the client side the proxy can be used to request synchronous or asynchronous calls to perform those operations. For asynchronous requests the client supplies an agent to be called back on receipt of the response from the server

```
class TTC_ASYNCHRONOUS_REQUEST
…
make (
    a_proxy: CHANNEL_PROXY
    an_argument_object: ANY
    an_expected_response_type: TTC_EIFFEL_TYPE
    a_response_procedure: PROCEDURE [ANY, TUPLE[ANY]) is…
```

The client can use the reflection mechanisms to determine the argument and response types, but it is not essential to do so since the client is obliged to say what type to use to unmarshal the response. The passed argument object will be checked for conformance when the call reaches the server, and when the response from the server is returned, the expected response type is used to check conformance of the response.

As long as the client stays running, the asynchronous RPC mechanism offers “at least once” delivery guarantees. This can be augmented by further guarantees no matter which side fails, by using patterns of persistent items either or both ends of the call.

Requests are only ever directed at “primary” processes. Secondaries merely get their persistent data updated see below.

Requests can have timeouts set, and asynchronous ones can be cancelled[2].

Persistent items, transactions and replication

It is persistence and optionally replication that provide the key to the reliability facilities of Eiflex. However, the obvious approach of storing the whole distributed component to persistent store at the end of each transaction is not really a viable solution on performance grounds, particularly in the high performance environments in which Eiflex has been deployed. Instead Channel servers have persistent items whose state (Eiffel objects associated with the data typing scheme mentioned above) is held in memory, and changes to it are written to transaction logs, and optionally sent to replica processes as transactions are terminated.

There is one key feature associated with persistent items. On the source (primary) side:

```
notify (term: ANY; an_update: TTC_CHANGE) is
    -- Notify `an_update'.
    -- It is assumed that the object already
    -- has the update applied
     require
    valid_term: my_type.has (term)
    primary: channel.is_primary
```

[2] As yet timeout and cancellation are primarily to prevent the client from being held up by missing or overloaded servers. The cancellations do not propagate to the server side. This is a desirable enhancement, but not essential one, since exactly once patterns can be constructed, and they can be used to ignore duplicates at the server side.

On the secondary side (which includes application initialization), the infrastructure applies the change to the persistent term to ensure the persistent terms have the same value as the primary last had (either through replication or roll forward after a crash). If and when this process becomes primary, the channel server "activate" feature is called at which point the channel server has to use its persistent state values to derive how to behave. Typically this would mean publishing all topics derived from the persistent state.

Topics and subscribers

Topics and persistent items have some behavioral similarities, in that in both cases the channel server is obliged to notify changes of value that can be applied remotely to keep cached copies in step. In the case of topics this feature is called publish, but otherwise it has the same signature as notify does in the persistent case. Unlike persistent items, topics are available for other eiflex components to subscribe to. A subscriber must effect 4 deferred features:

```
on_term (a_topic: TTC_TOPIC) is
            -- Initial value has arrived

on_type_invalid (a_topic: TTC_TOPIC) is
            -- Proxy and server types don't conform.

on_txn (a_bundle: TTC_CHANGE_BUNDLE) is
            -- Changes after server txn completed.

on_up_to_date (a_channel: TTC_CHANNEL) is
            -- implies connection to server established

on_doubtful (a_channel: TTC_CHANNEL) is
            -- Connection to the server has been lost
```

Changes of state

When a transaction is undertaken, persistent objects and topics will invariably undergo changes of state. Rather than persist the whole item or publish the whole topic value, the channel server is obliged to supply a "change" object. That is one which when applied to the "old" value will result in the "new" value. For example if the state is a hash table, then the change might be an addition or a removal. It is these lightweight change objects which are written to persistent store, or broadcast to topic subscribers.

In a subscriber, there is an obvious contract that the change applied to the old cached value results in an equal value to that held by the server topic. However without sending the whole server topic value with each change, the contract cannot be checked, and sending the whole value defeats the performance objective of delivering lightweight change objects. Equally the subscriber cannot query the topic value in the assertion, since the whole system is asynchronous, and the server may have moved onto a later transaction. So this contract is at present just a comment.

Reliable message streams
Messages streams have some similarities to request / operation call if they go from client to server, and are similar to publish / subscribe with topics if they go from server to client. They differ in one key aspect. Message streams offer a guarantee of exactly one delivery of each and every message.

To achieve the guarantees the message stream sender makes use of history files and both sender and receiver use persistence to know where they are up to. There is a negotiation phase on connection to synchronize and resend any missing messages.

The sender (client or server) can cause a new message series to be started, at which point previous series can no long be sent – although may optionally be retrieved, unless they have been "archived".

Reflection
For reflection purposes every channel server publishes topics giving the list of operations, topics, persistent items, message sources and message sinks in that channel server.

Exceptions
Developer exceptions are sometimes used to report distribution failures and type mismatches. Some of these may result from problems that might be overcome with time – e.g. an attempt to perform a synchronous call to a channel when a "primary" server process containing the channel cannot be found. Some may represent programming errors – for example an attempt to call an operation that does not exist, but the channel server does.

This style of exception use differs from the usual broken contract convention adopted by the Eiffel method, but is a natural consequence of the need to be tolerant to distribution failures.

3 Evolution of Eiflex

The first incarnation of Eiflex was single threaded, used CORBA for all the distribution interactions, did not have reliable messages streams, its distributed objects were much coarser grained, and it used a more restricted data typing scheme. It did however enable prewritten data manipulation components to be wired together with configuration diagrams to achieve business functionality without necessarily writing Eiffel code.

The current version has evolved a long way from that - which never-the-less is still in use at the Chicago Board of Trade for one of its applications.

The desire for a multithreaded version was prompted by the need to achieve more parallelism for I/O and business logic. In the single threaded version all IO was based on polling with timers, and the pipelining throughput suffered badly because of it.

CORBA was discarded for a number of reasons:

- To achieve the publish subscribe mechanisms involved the use of an intermediate service (in our case we used the Event Service), adding to the management overheads.

- The CORBA data typing was not flexible enough to carry the sorts of data we needed, and so we were mapping to strings, and about the only argument type we used was string. Object by Value was emerging, but it was not really a viable alternative.
- The particular CORBA supplier we were using did not offer any hot standby mechanisms, so we had to adopt different reliability policies for the CORBA service processes from the Eiflex processes.
- The threading policy from the supplier we were using was to invoke a thread per call, rather than the consumer queue approach we had already decided was how we wanted to schedule work through the distributed objects.
- It was RPC oriented, and the publish subscribe and reliable messaging interactions all had to be implemented with RPC behind the scenes, making them much less efficient that they should have been.

The behaviour neutral data typing was also dropped, and values based on Eiffel objects adopted instead. This was a complete reversal of the original architecture. The wiring scheme was flexible, but it was inefficient, and was more like writing data flow diagrams than OO programming, and most of the programmers who were exposed to it did not like it. They were much happier writing OO Eiffel code.

A separate short paper submitted to this conference describes why SCOOP was not used to implement the later versions of Eiflex, but essentially the main reasons were – it didn't exist, it would deadlock in too many situations, and we suspect it might not scale up well.

4 Some implementation details

The Channel library summarized earlier is all a programmer using Eiflex needs to be aware of. However under the hood, there is much going on, and this section describes how some of the key features of the reliable distribution are actually achieved.

4.1 Buffered Consumer Threads (SCOOP processors)

Channel proxies and channel servers reside in buffered consumer threads. These are similar in concept to a SCOOP processor. The thread has a queue of channel commands, each one corresponding roughly to one of the arrows in the earlier proxy / server interaction picture, with the addition of a command for the execution of timed events. The commands are executed in turn (except when in the waiting state see below) on the proxies or servers that were created in that thread. Every channel server or channel proxy object becomes adopted by the consumer thread in which it is created. So within a channel server, or client of a proxy, the code can assume all the usual rules of the Eiffel method. The programmer is coding within a shielded single threaded world. Of course there is still the issue of avoiding deadlock if synchronous RPC is used. Which leads to the next subject.

Avoiding deadlock - Logical threads

There is no reservation mechanism between Eiflex channels, and command query separation can not be relied upon between consecutive distributed operation calls. If one transaction requires the exclusive synchronous use of more than one channel server resource, then careful application design is required to implement application level reservation. This implies that solutions to problems such as that of the dining philosophers are not as elegant in Eiflex as the SCOOP solution. However in practice this is not a significant issue in the sorts of domain Eiflex is targeted at.

We should emphasize again that given the reliability guarantees offered by Eiflex, and the existence of persistent intermediate state, most interaction between distributed Eiflex components is expected to be explicitly asynchronous and pipelined. However Eiflex does provide synchronous RPC, and of course this can result in deadlock. To help avoid some potential deadlocks, Eiflex provides a feature (that we call "logical threads") that we believe reduces the risk of deadlock in most real life situations. Essentially the feature ensures that if a sequence of calls between objects works as a single threaded application, then distributing the objects will not cause that sequence to deadlock.

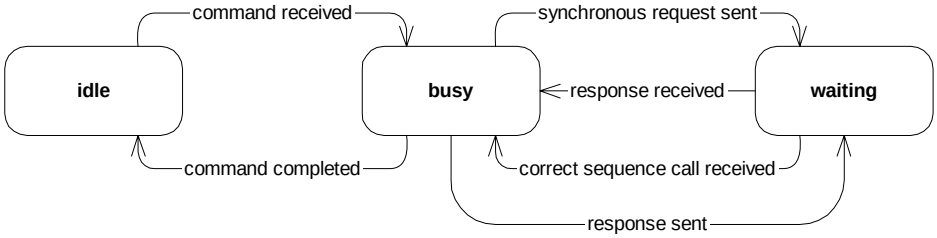


Fig. 3. State diagram for the buffered consumer threads

The temporal example (fig 4) illustrates how some synchronous calls that arrive at a consumer thread are executed as soon as they get to the head of the queue (the idle <-> busy transformation), but some leap the queue (the waiting -> busy transformation).

The most important aspects of the diagram are what happens at times 3, 4, 6/7 and 11. At time 3, when Thread1 issues the synchronous request to call Ch3, Thread1 moves into the waiting state. In this state the thread is still accepting commands for execution and putting them in its queue. So for example at time 4 when another synchronous call arrives for Ch2, the history of that call shows that it did not arise as a direct consequence of the outgoing call at 3, so that incoming call is queued. However at point 6, another incoming synchronous call arrives, and the history of that call shows that it did arrive as a direct consequence of the call at 3, and so that call leaps to the front of the queue and is executed, putting thread1 back into the busy state. When finally the response to the original call is returned at time 10, Thread1 returns to the idle state, and pops the next command off its queue which in this example is the call that arrived at time 4.

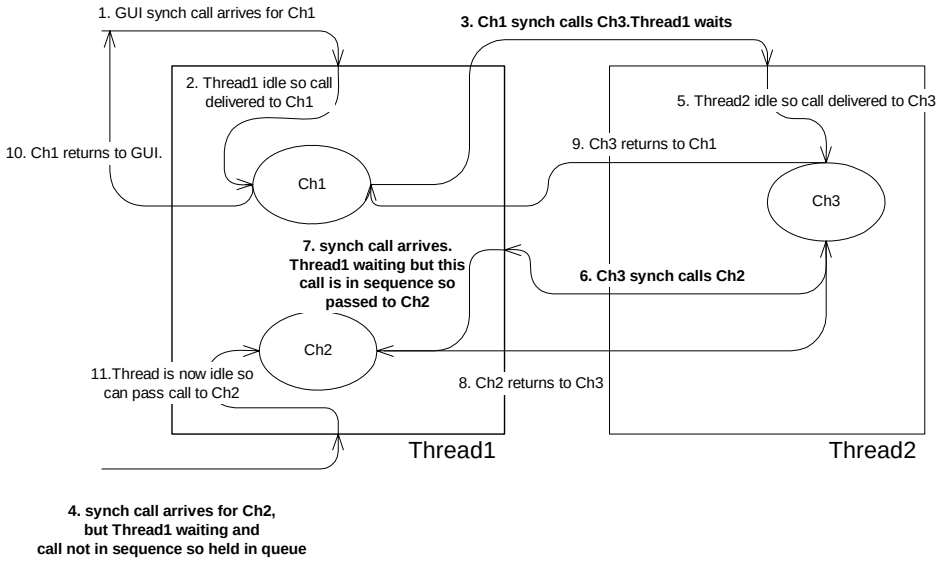


Fig. 4. Temporal example of Logical threads – the technique used to reduce the likelihood of deadlock

We call the sequence 1-3,5-10 a logical thread, and the sequence 4,11 is another. It is our contention that although this feature does not provide a solution to all deadlocks (and does not really help the dining philosopher's problem), it does avoid many potential deadlock situation where calls traverse up and down an object hierarchy.

4,2 Processes and Channel Tunnels

The consumer threads reside inside operating system processes, and Eiflex processes are interconnected through TCP based connections that carry the RPC, publish / subscribe, messaging and replication protocols. As was mentioned above, we call the inter connection between channel proxy and channel server a channel tunnel, but actually a channel tunnel multiplexes all the interconnected channels between processes.

The remote tunnels (connecting processes) exploit 3 threads on the client side and 2 threads on the server side of the boundary. At the client end a connection thread is responsible for finding and establishing the connection to the remote end. Once its task is complete, it dies and 2 further threads are created – one for synchronous reading and one for synchronous writing. The connection thread is resurrected should the link become broken for any reason. Using a dedicated thread for synchronously reading and writing greatly simplifies the socket handling and delegates scheduling issues to the operating system, whose writers are far more adept at such things. (We

should indicate that our comparisons of Windows and Solaris suggest that Unix is far better at handling blocking sockets than Windows is).

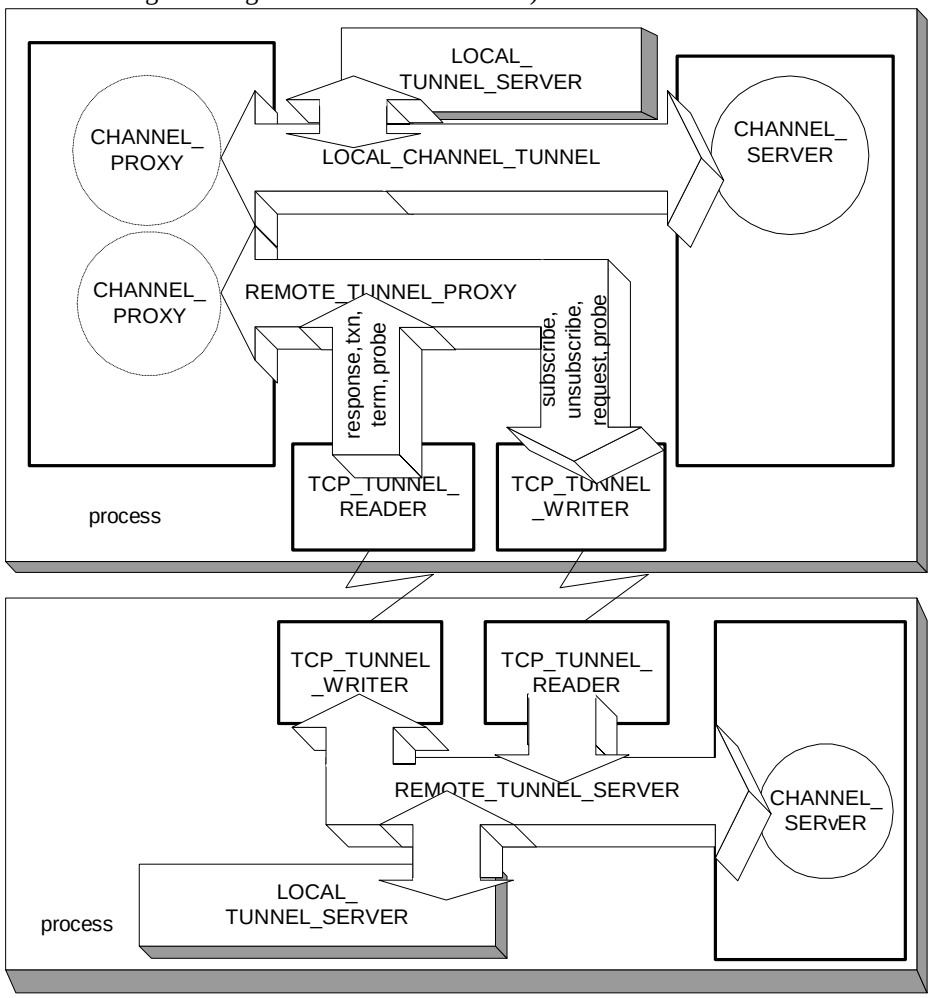


Fig. 5. Tunnel interconnect between Eiflex processes

4.3 Server processes, the Bootstrap Server registry, and Node Managers

As part of the fault tolerance attributes, one prime objective of an Eiflex system is that it be self managing, and require little or no human intervention once it has been set up, and certainly no human intervention to achieve recovery.

The number of and physical locations of the server processes in a particular Eiflex deployment are expected to be relatively static. However the number of channels and

consumer threads is expected to be dynamic. Eiflex has a “well known” bootstrap channel server, the registry (not to be confused with the Windows concept with the same name), whose role in life is to publish the location of the server processes, and to publish in which server processes the dynamic channel servers reside. The former i.e. the location of the server processes in the system, is expected to be set up by humans. The latter is very dynamic, and as was mentioned before, the channel servers register themselves with this registry channel on creation and at other important times (For example should the registry disappear for some reason and be resurrected, all processes correlate what the registry should know about them with what it does publish about them, and then corrects any discrepancies).

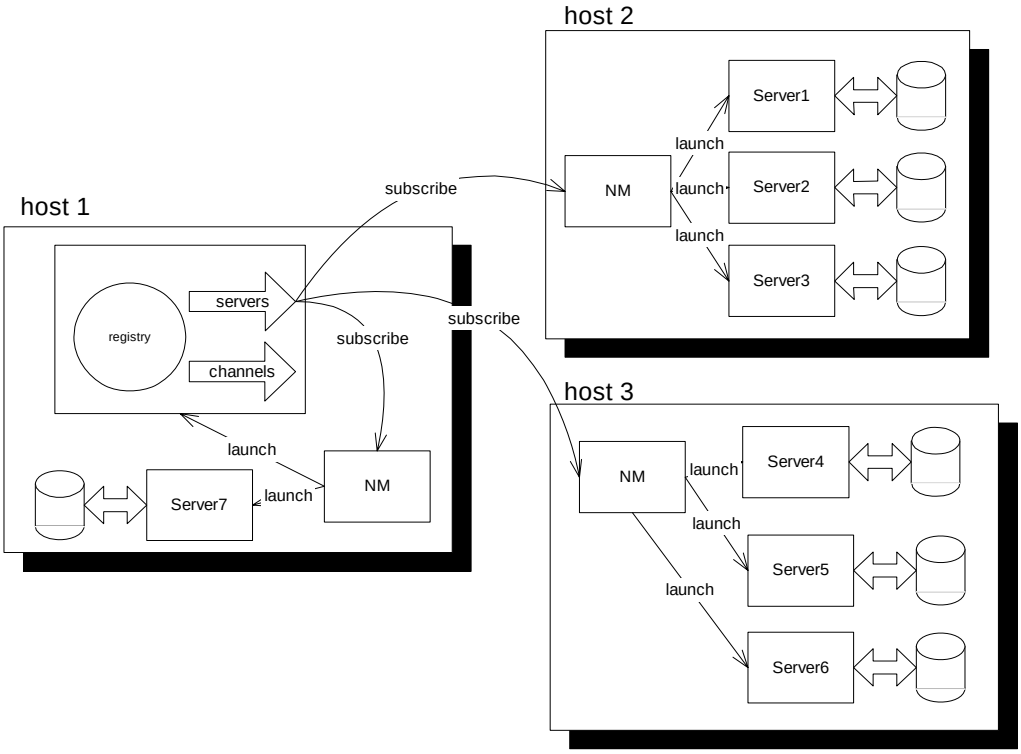


Fig. 6. Node manager the watchdog process driven by the registry. The registry publishes what should be running, the NMs make sure it is.

At first sight this registry channel server might appear to be the weak link in the reliability story. But remember, this is a channel server like any other so can itself have a hot standby, but even if only a single instance is running, it publishes its information as topics, and so all the other processes have cached copies of the

registration information, and if the registry dies, these cached copies can still serve the find and bind mechanisms pending the node manager restarting the registry.

So this registry channel is just a normal channel like any other, and only differs in that it is a pre-supplied channel server, its location(s) needs to be specified as a bootstrap option to each process in the "system", and it is used by Eiflex itself in the find and bind mechanisms of joining a channel proxy to a channel server. This registry channel publishes 3 main topics:

- Server processes – relatively static definition of which processes reside on which boxes, the location of the externalized state files for that server, and the executable that runs that server.
- Channels – dynamic definition of extant channels.
- Schedule – similar to Unix cron, defines what processes should be running when, and also when they should be checkpointed.

The Node Manager is a special watchdog client process supplied as part of Eiflex. One copy runs on each hardware box, and it uses the registry "servers" topic to start, stop and watchdog the processes that should run on its own box according to the Schedule topic. Starting node managers is a platform specific task – for example a service on Windows – or a daemon on Unix, but this is the only area where a platform characteristic shows through in managing an Eiflex deployment.

4.4 Configuration files, Factory, Persistence and Replication

Each server process normally starts up with a set of configuration files. These represent the state of all channel servers in the process at the time the process was last checkpointed, together with transaction logs holding all the changes of state since then. The thread and channel server factories are invoked at load time to create the threads and channel objects from the externalized specifications in these files. Once the channel servers are created, they their persistent objects are given their state as of the last completed transaction, and then if this process is to be the "primary" the channel servers are "activated".

As transactions are performed changes of state are written to transaction logs which can then be used to roll forward should the process break and be resurrected for any reason. The server processes can also be "checkpointed", at which point the total persistent state of the process is written out to its configuration files, and a new transaction log started.

Server processes have "replication ids" and multiple servers with the same replication ID are considered to be replicas. A negotiation phase determines which will be primary at any time. (Client channel proxies only ever talk to channel servers on primary server processes).

5 Summary

This paper posits the suggestion that when providing distributed solutions it is essential to consider the consequences of distribution failures. It may be acceptable in

some circumstances to have a complete system fail if any part of it does, however if that is not an acceptable outcome, then for certain types of application domain – particularly enterprise integration, Eiflex offers a framework for survival.

Eiflex has evolved from its initial architecture, largely driven by the demands of the application domains in which is has been used. There are clearly issues in its architecture which could still be improved, and there are clearly some application domains where it would not be a suitable candidate. Its success at the Chicago Board of Trade, and at OM Jiway demonstrates that the technology (albeit earlier versions) is a viable commercial scale middleware.

6 Reliable dining philosophers

Having said that the architecture is not designed with the intent of solving the dining philosophers problem, here is the essence of the Eiflex solution. It is worth noting the use of asynchronous timers. Depending on the deployment decisions taken, each fork and philosopher may share a thread with other forks and or philosophers – they cannot synchronously wait and prevent the other channel servers in the same thread from running.

Although not anything like as trivial a solution as that presented in SCOOP chapter of [OOSC], we all know that behind the scenes the SCOOP runtime reservation mechanism is frantically doing the same sort of reserve, backoff, try again negotiation. One clear advantage of the Eiflex approach is that the forks and philosophers can independently crash, but since the reservation state is persistent in the forks, then they will come back (either a hot standby if replication is used, or a restart of the same process if replication is not in use) consistent with all the other surviving components, and the system will carry on. For this particular application it is only necessary for the fork to hold any persistence, and that is the id of the philosopher that currently has it reserved. Another advantage of this Eiflex approach is that the philosophers publish their state, including hungry and starving states if they get to wait progressively longer for their food. GUI clients for example can readily subscribe to this topic and present the state of each philosopher.

```
class  FORK
inherit
    TTC_CHANNEL_SERVER
       rename
          make as cs_make
       end
    PHILOSOPHY_CONSTANTS
       export {NONE} all end
Create
    make
feature {NONE} -- intialization

    make (an_id: PATHNAME_ID) is
       local
          l_operation: TTC_OPERATION
       do
          cs_make (an_id)
          start_txn
          create l_operation.make (
             Reserve_operation_name,
```

```
            Eiffel_pathname_id_type,
            Eiffel_boolean_type,
            agent reserve)
         add_operation (l_operation)
         create l_operation.make (
            Release_operation_name,
            Eiffel_pathname_id_type,
            Eiffel_none_type,
            agent release)
         add_operation (l_operation)
         create persistent_reservation.make (
            Persistent_reservation_name,
            Eiffel_pathname_id_type,
            Void)
         add_persistent
         terminate_txn
      end

feature {NONE} -- operation agents

   reserve (a_pid: PATHNAME_ID): BOOLEAN_REF is
      do
         if not reserved then
            persistent_reservation.notify (
               a_pid,
               create {TTC_REPLACE_UPDATE [PATHNAME_ID]}.
                  make (a_pid))
         end
         Result := persistent_reservation.item.is_equal (a_pid)
      end

   release (a_pid: PATHNAME_ID): NONE is
      do
         if persistent_reservation.item.is_equal (a_pid) then
            persistent_reservation.notify (
               Void,
               create {TTC_REPLACE_UPDATE [PATHNAME_ID]}.make (Void))
         end
      end

feature {NONE} -- implementation

   reserved: BOOLEAN is
      do
         Result := persistent_reservation.item /= Void
      end

   persistent_reservation: PERSISTENT [PATHNAME_ID]

end

class PHILOSOPHER
inherit
   TTC_CHANNEL_SERVER
      rename
         make as cs_make,
         activate as release_and_think
      redefine
         release_and_think
      end
   PHILOSOPHY_CONSTANTS
      export {NONE} all end
create
   make
```

```
feature {NONE} -- initialization

    make (
          an_id: PATHNAME_ID
          a_left_fork: TTC_CHANNEL_PROXY
          a_right_fork: TTC_CHANNEL_PROXY) is
       do
          cs_make (an_id)
                -- Since adding topics invloves the value of
                -- the "topics" topic being published, this
                -- must be performed in an explicit transaction
          start_txn
          create state_topic.make (
             State_topic_name,
             Eiffel_string_type,
             "initializing")
          add_topic (state_topic)
          left_fork := a_left_fork
          right_fork := a_right_fork
          create hungry_timer.make (agent try_to_eat, 10)
          create thinking_timer.make (agent try_to_eat, 500)
          create eating_timer.make (agent release_and_think, 1000)
          create starving_timer.make (agent starve, 50000)
          terminate_txn
       end

feature {NONE} -- implementation

    hungry_timer: SEPARATE_RELATIVE_TIMER
    thinking_timer: SEPARATE_RELATIVE_TIMER
    eating_timer: SEPARATE_RELATIVE_TIMER
    starving_timer: SEPARATE_RELATIVE_TIMER
    left_fork: TTC_CHANNEL_PROXY
    right_fork: TTC_CHANNEL_PROXY
    State_topic: TTC_TOPIC_SERVER


    think is
       require
          all_timers_stopped: all_timers_stopped
       do
          publish_state (Thinking)
          thinking_timer.start -- will call try_to_eat on expiry
       ensure
          thinking: thinking_timer.is_running
       end


    eat is
       require
          all_timers_stopped: all_timers_stopped
       do
          publish_state (Eating)
          eating_timer.start -- will call release_and_think on expiry
       ensure
          is_eating: eating_timer.is_running
       end


    starve is
       require
          am_hungry: hungry_timer.is_running
       do
          publish_state (Starving)
       ensure
          still_hungry: hungry_timer.is_running
       end
```

```
try_to_eat is
   require
      not_still_thinking: thinking_timer.is_stopped
      not_still_eating: eating_timer.is_stopped
   local
      l_current_state: STRING
      l_rescued: BOOLEAN
   do
      if not l_rescued then
         if reserve (left_fork) then
            if reserve (right_fork) then
               if starving_timer.is_running then
                  starving_timer.stop
               end
               eat
            else
               release (left_fork)
            end
         end
         if eating_timer.is_stopped then -- not eating
            if starving_timer.is_stopped then -- not starving
               starving_timer.start
               l_current_state ?= State_topic.term
               if
                  not (l_current_state.is_equal (Starving) or
                  l_current_state.is_equal (Hungry))
               then
                  publish_state (Hungry)
               end
            end
            hungry_timer.start
         end
      else
         hungry_timer.start
      end
   ensure
      is_eating_or_hungry:
         eating_timer.is_running or else hungry_timer.is_running
   rescue
      -- may have lost contact with forks
      l_rescued := true
      retry
   end

release_and_think is
      -- On activation this is the first state we enter
      -- even after recovery. So first we must tidy up the
      -- reservations. This is perhaps a bit of a glitch after
      -- recovery, since the previous incarnation may have been
      -- eating or even hungry when it broke, but this is the
      -- simplest restart point as long as we get the forks into
      -- the right state.
   require else
      all_timers_stopped: all_timers_stopped
   local
      l_rescued: BOOLEAN
   do
      if not l_rescued then
         release (left_fork)
         release (right_fork)
      else
         eating_timer.start
      end
      think
   rescue
      -- may have lost contact with forks
      l_rescued := true
```

```
            retry
        end

    reserve (a_fork: TTC_CHANNEL_PROXY): BOOLEAN is
        local
            l_request: TTC_SYNCHRONOUS_REQUEST
            l_response: BOOLEAN_REF
        do
            create l_request.make (
                a_fork,
                Reserve_operation_name,
                Eiffel_boolean_type,
                id)
            l_request.execute
            l_response ?= l_request.response
            Result := l_response.item
        end

    release (a_fork: TTC_CHANNEL_PROXY) is
        local
            l_request: TTC_SYNCHRONOUS_REQUEST
            l_response: ANY
        do
            create l_request.make (
                a_fork,
                Release_operation_name,
                Eiffel_none_type,
                id)
            l_request.execute
            l_response ?= l_request.response
        end

    all_timers_stopped: BOOLEAN is
        do
            Result :=
                hungry_timer.is_stopped and
                thinking_timer.is_stopped and
                eating_timer.is_stopped and
                starving_timer.is_stopped
        end

    publish_state (a_state: STRING) is
        do
            io.put_string (id.as_string + " " + a_state + "%N")
            State_topic.publish (
                a_state,
                create {TTC_REPLACE_UPDATE [STRING]}.make (a_state))
        end
end
```

Acknowledgements

Gordon Jones and Emmanuel Bouyer have been responsible for the Eiflex middleware in recent years, and in particular during its conversion from single threaded using CORBA to multithreaded without. However its first version involved a number of other authors. We are indebted in particular to Martin Coen and Neil Lancaster for much of the distribution architecture, and to Eric Bezault for the essential aspects of the meta typing scheme, and of course the gobo tools.

We are of course enormously indebted to Bertrand Meyer for Eiffel itself, and to Eiffel Software for providing us with an excellent platform independent thread cluster.

References

[OOSC] B. Meyer, Object-Oriented Software Construction, Second Edition, Prentice Hall, ISBN 0-13-629155-4, 1997.
[ETL3] B.Meyer, Eiffel the Language – third edition. Work in progress accessed via Bertrand Meyer's home page - http://se.ethz.ch/~meyer, 2006.
[ECMA] Eiffel language standard. 2005 - http://www.ecma-international.org/publications/standards/Ecma-367.htm
[OMG] CORBA and related standards, www.omg.org
[OMC] OM Click, http://www.omxgroup.com/omxcorp/ourbusiness/technologysolutions/oursystems/clickxt/
[FIX] The Fix Protocol, http://www.fixprotocol.org

An Alternative Model of Concurrency for Eiffel

Phillip J. Brooke[1] and Richard F. Paige[2]

[1] School of Computing, University of Teesside
Middlesbrough, TS1 3BA, U.K. P.J.Brooke@tees.ac.uk
[2] Department of Computer Science, University of York
Heslington, York, YO10 5DD, U.K. paige@cs.york.ac.uk

Abstract. The Simple Concurrent Object-Oriented Programming (SCOOP) is the leading proposed mechanism for introducing concurrency to Eiffel. We outline a number of concerns, related to the semantics of SCOOP, and present an alternative concurrency model for Eiffel that alleviates many of these problems. Our alternative model aims to preserve the existing behaviour of sequential programs and libraries whereever possible. Comparison with the SCOOP model is made, and ongoing work on evaluating the alternative model is discussed. A sketch of the model in CSP is given.

1 Introduction

The Simple Concurrent Object-Oriented Programming (SCOOP) [10, 11] mechanism is proposed as a way to introduce inter-object concurrency into the Eiffel programming language [10, 6]. SCOOP extends the Eiffel language by adding one keyword, **separate**, which can be applied to classes, entities, and formal routine arguments. Application of **separate** to a class indicates that objects of that class executing in their own (conceptual) 'thread' of control; application of **separate** to entities or arguments of routines indicate that these constructs are points of synchronisation.

1.1 Aim of this work

As part of other work constructing a CSP [9] model of the interesting interactions of SCOOP programs [2], we noted some complications or ambiguities with SCOOP in its current form. This paper proposes an alternative formulation of concurrency for Eiffel, which alleviates many of these concerns.

1.2 Scope and limitations

The aim of this work is to build an alternative model for concurrency in Eiffel. We take SCOOP as it is currently documented [10, 11].

1.3 Related work

An incomplete prototype of the SCOOP mechanism was implemented by Compton [5] by building upon the GNU SmartEiffel compiler and run-time system. A prototype preprocessor implementation was constructed by Fuks et al. for ISE Eiffel [8].

More recently, Nienaltowski et al. [12] have produced the most complete implementation of SCOOP to date. This (in common with other prototypes) is a preprocessor that rewrites SCOOP-using classes. None of these prototypes can be considered a full implementation of the SCOOP specification in [10].

One reason for the incomplete nature of the prototypes is that the SCOOP mechanism and its underlying semantics is both complex to understand and difficult to implement in a compiler and run-time environment. The complexities inherent in the interactions between the language and the implicit, underlying run-time system are potentially confusing.

1.4 Overview

In Sections 2 and 3, we outline Eiffel and SCOOP. We summarise our critique of SCOOP in Section 4; a more detailed critique is in the companion paper to this [3].

We present our alternative model in Section 5, along with a CSP sketch in Section 7, and discuss our contribution in Section 8 before concluding in Section 9.

2 Eiffel

Eiffel is an object-oriented programming language and method [10, 6]; it provides constructs typical of the object-oriented paradigm, including classes, objects, inheritance, associations, composite ("expanded") types, generic (parameterised) types, polymorphism and dynamic binding, and automatic memory management.

A short example of an Eiffel class is shown in Figure 1. The class $CITIZEN$ inherits from $PERSON$ (thus defining a subtyping relationship). It provides several attributes, e.g., $spouse$ and $children$, which are of reference type (in other words, $spouse$ refers to an object of type $CITIZEN$); these features are publicly accessible (i.e., are exported to ANY client).

Attributes are by default of reference type; a reference attribute either points at an object on the heap, or is $Void$. The class provides one expanded attribute, $blood\_type$. Expanded attributes are also known as composite attributes; they are not references, and memory is allocated for expanded attributes when memory is allocated for the enclosing object.

The remaining features of the class are routines, i.e., functions (like $single$, which returns $true$ iff the citizen has no spouse) and procedures (like $divorce$, which changes the state of the object).

```
class CITIZEN inherit PERSON
feature {ANY}

  spouse: CITIZEN
  children, parents: SET[CITIZEN]
  blood_type: expanded BLOOD_TYPE

  single: BOOLEAN is
    do Result := (spouse=Void)
    ensure Result = (spouse=Void)
    end

feature {BIG_GOVERNMENT}

  marry is ...
  have_child is ...
  divorce is
    require not single
    do ...
    ensure single and (old spouse).single
    end

invariant
  single or spouse.spouse = Current;
  parents.count <= 2;
  children. for_all ((c:CITIZEN):BOOLEAN do
    Result := c.parents.has(Current) end)
end -- CITIZEN
```

Fig. 1. Eiffel class interface

These features may have preconditions (**require** clauses) and postconditions (**ensure** clauses). The former must be true when a routine is called (i.e., it is established by the caller) while the latter must be true when the routine's execution terminates.

Finally, the class has an invariant, specifying properties that must be true of all objects of the class at stable points in time, i.e., before any valid client call on the object.

For more details on the language, see [10] or the more recent [6].

3 SCOOP

3.1 Outline

SCOOP introduces concurrency to Eiffel by addition of the keyword **separate**; it is the responsibility of the underlying run-time system and compiler to deal with the subtle (and, in some cases, complicated) semantic problems introduced by the addition. The **separate** keyword may be attached to the definition of a class, the declaration of an entity or formal routine argument. Examples of these three types of attachments are as follows:

separate class ROOT
x: **separate** PROCESS
f(y:**separate** PROCESS)

An object created as **separate** has its own conceptual thread of control (although this is complicated shortly when we discuss 'processors').

Access to a **separate** object, whether via an entity (e.g., x in our example above) or formal argument (e.g., y) indicates different semantics to the usual sequential Eiffel model. In the sequential model, feature calls cause execution to switch to the called object, whereupon the feature executes, and (perhaps after storing a result), execution continues at the next instruction.

In SCOOP, routine calls to x or y are asynchronous. The called object can queue multiple calls in a FIFO, allowing callers to continue concurrent execution.

Function calls and reference to attributes are synchronous — but may be subject to lazy evaluation. This lazy evaluation is a form of Caromel's *wait-by-necessity* mechanism [4].

Additionally, races are prevented by the convention that a **separate** formal argument causes the object to be exclusively locked during that feature call. This locking is known as *reservation* in SCOOP.

3.2 Processors

SCOOP introduces the notion of a *processor*. When a **separate** object is created, a new processor is also created to *handle* its processing. This processor is called the object's *handler*.

Thus, a processor is an autonomous thread of control capable of supporting *sequential* instruction execution [10]. A system in general may have many processors associated with it.

Compton [5] introduces the notion of a *subsystem* —a model of a processor and the set of objects it operates on— to distinguish the execution of sequential and concurrent programs. In his terminology, a **separate** object is any object that is in a different subsystem. In this paper, we will refer to subsystems rather than processors (to avoid possible confusion with real CPUs).

3.3 Preconditions and waiting

Eiffel uses **require** and **ensure** clauses for specifying the pre- and postcondition of features. In sequential programming, a **require** clause specifies conditions that must be established and checked by the client of the routine; the **ensure** clause specifies conditions on the implementer of the routine. If a precondition evaluates to false, an exception is raised.

In SCOOP, a **require** clause on a feature belonging to a **separate** object specifies a *wait* condition: if feature's **require** clause evaluates to false, the processor associated with that object waits the precondition is true before proceeding with feature execution.

SCOOP uses the precondition as a *guard*. The intent of this mechanism is that another object may call routines on x causing the wait condition to evaluate to true. This also admits that the entire system may become deadlocked: the run-time system has a duty to detect such circumstances.

4 Problems with SCOOP

As it is currently understood, SCOOP suffers from under-specification and a number of complications or ambiguities [3]:

1. Chains of calls could cause deadlock unless reservations can be 'passed-on'.
2. It is unclear when reservations should be released.
3. Although SCOOP offers high parallelism, the use of subsystems to group objects and implicit reservations reduce this.
4. It is unclear when blocked preconditions should be rescheduled, particularly when there is contention for a single resource.
5. Priorities and call queues interact badly.
6. The formulation of separate-ness adds complication: the layers of objects, handlers (subsystems) and systems; the necessity for rules to prevent traitors; and the overloading of formal arguments with implicit reservation make it difficult to pass separate parameters without locking.
7. Some areas are omitted:
   (a) (asynchronous) exceptions, which are needed for duels;
   (b) real-time; and
   (c) interrupts (from external processes and devices).

8. Implementation matters:
   (a) failure of the underlying communication system;
   (b) races on reservations;
   (c) termination detection;
   (d) deadlock detection; and
   (e) scalability.

More details are given in a companion paper [3]. The overall complexity of SCOOP makes it particularly challenging to both implement and model.

5 An alternative model

We now present our alternative concurrency model for Eiffel. We start from (sequential) Eiffel, rather than from SCOOP.

We want existing (sequential) programs to have the same behaviour as now *without changing the text of the program*. Additionally, we want to preserve as much use of existing libraries even when used in a concurrent context.

1. Each object has a notional thread of control. (This includes expanded objects.)
2. All objects have a queue of feature calls made by other objects. These calls are processed in FIFO order. At most one feature call can be executing on an object at any one time.
   Calls from the object to itself are handled immediately.
3. *All calls are synchronous unless* the text of the program indicates otherwise using the **async** keyword to qualify the call, e.g., **async** $a.f$. Thus concurrency is introduced explicitly in the program text, ensuring that existing programs do not inadvertently introduce concurrency. We speculate that real programs (rather than illustrations) will generally use more sequential calls than asynchronous calls, so the programmer burden is lower.
   A compiler can be optimistic and treat synchronous calls as asynchronous if it can guarantee semantically equivalent behaviour.
   Function calls result in *two* entries being enqueued: one entry to indicate the feature called (and its parameters, etc.) and a second to indicate the synchronisation point where it requires the result of that function. This could be later in the feature body, using the wait-by-necessity mechanism, or even omitted entirely if the result is not actually needed (although the developer might wish to question this).
   Procedure calls only require one entry to be enqueued. Reference to attributes should be treated as for function calls.
   Creation procedures can also be decorated with **async**.
   We disallow the use of **async** on an unqualified call, since we do not (yet) address multiple threads of control in an object.
4. Feature calls can only be made on locked objects, i.e., *Current* holds the 'active lock' on the callee.

5. All objects given as formal arguments are locked unless explicitly excluded using the **unlocked** keyword in the argument list, h (a: **unlocked** C). Locking is the default to reduce potential races when existing sequential libraries are re-used in a concurrent context.
6. 'Lazy locks' are automatically made on unlocked objects that are the subject of a feature call. This lock only exists for the duration of that call. This applies to the body of features and also to assertion evaluation. Lazy locks can apply to both synchronous and asynchronous calls; access to an asynchronous result requires another lazy lock against the object.
   The main purpose of this clause is to allow existing sequential code to work unchanged, even though an 'active lock' is required to enqueue feature calls on other objects. Additionally, it allows blocks of code that do not require mutual exclusion on a object can still use it (when no other object is reserving it). This is at the expense of potentially greater overheads.
   Nienaltowski points outs that "acquiring a lazy lock corresponds to calling an implicit enclosing routine that wraps the actual call."[3] Thus an alternative formulation based on examining a feature body for unlocked calls might be possible, although we need to handle the case where loops or other calculations make it difficult to determine the targets of these calls.
7. Locks are passed on through call chains, as described in [2]. Thus an object may be locked if
   - it is not currently locked; or
   - a parent caller (whether direct or indirect) holds the lock. While a child call holds the lock, the parent temporarily loses the active lock.

   See the paper describing a CSP model of SCOOP for further details [2].
8. Features have wait conditions as for SCOOP.
   Wait conditions apply to all calls, both synchronous and asynchronous. Calls that cannot obtain all locks needed to proceed, or which fail their wait conditions are suspended and retry later.
   Suspended calls are queued and should be re-evaluated whenever the system detects a change such that the wait conditions may now be true. To ensure liveness, suspended calls should be re-attemped in FIFO order of suspension, although a given call might still be blocked on other locks or wait conditions, or there may be other features active on that object.
   If the compiler or run-time system can determine that all wait conditions can never be true, then an error or exception should result. In particular, if at the time the call is enqueued, it can be determined that the wait conditions will never be true, then a synchronous (precondition violation) exception should be raised.
   Future work (dealing with real-time programming) may impose or suggest further scheduling policies.
9. The developer should choose whether (groups of) asynchronous exceptions[4] cause an object to either

[3] As part of Nienaltoski's attributed review.

[4] A paper in preparation, *Exceptions in concurrent Eiffel.* deals with this part of the proposal in more detail. Contact the authors for draft copies.

- cause the whole system to halt; or
- cause the object to die without attempting to process any other feature calls from its queue. Future attempts to access the object cause the caller to receive the exception separate_object_failure.

10. Objects are assigned to processing resources (e.g., dedicated CPUs, POSIX threads) that we call *partitions*. This assignment is via a policy set by the engineer at compile time. There is no intrinsic reason why objects should not be able to migrate (or indeed, be replicated for fault-tolerance) between partitions. However, we do not describe that mechanism here.
11. A problem in the underlying partition communication system that prevents communication with an object results in the exception 'partition_communication_failure'.
12. Interrupts from outside the Eiffel system (e.g., operating system or device interrupts) are placed in a queue that can be waited on, or interrogated by other objects.

6 Example program: buffer-consumer

6.1 SCOOP version

We first present a standard SCOOP version of the buffer-consumer example, directly derived from http://www.cs.yorku.ca/~jonathan/students/FuksSlides.ppt and [11]:

```
class ROOT_CLASS creation
        make
feature

        b: separate BUFFER[INTEGER]
        p: PRODUCER
        c: CONSUMER

        make is
                do
                        create b.make
                        create p.make(b)
                        create c.make(b)
                end
end -- ROOT_CLASS

separate class PRODUCER creation
        make
feature
        buffer: separate BUFFER[INTEGER]

        make (b: separate BUFFER[INTEGER]) is do buffer := b; keep_producing end

        keep_producing is
```

```
        do
                ...
                store_in_buffer (buffer, i)
                ...
        end

        store_in_buffer  (b : separate BUFFER[INTEGER]; i:INTEGER) is
                require
                        not b. is_full
                do
                        b.put(i)
                ensure
                        b.has(i)
                end
end -- PRODUCER

separate class  CONSUMER creation
        make
feature
        buffer:  separate BUFFER[INTEGER]

        make (b: separate BUFFER[INTEGER]) is do buffer := b; keep_consuming end

        keep_consuming is
        local
                ...
        do
                ...
                i  :=  consume_from_buffer(buffer)
                ...
        end

        consume_from_buffer (b: separate BUFFER[INTEGER]) : INTEGER is
                require
                        not b.is_empty
                do
                        Result := b.item
                        b.remove
                ensure
                        b.count = old b.count - 1
                end

end -- CONSUMER
```

We assume that we have a standard buffer:

```
class  BUFFER[G] creation
        make

feature
```

```
        put (x:G) is
                require
                        not  is_full
                ensure
                        count = old count + 1
                end

        item : G is
                require
                        not is_empty
                end

        remove is
                require
                        not is_empty
                ensure
                        count = old count − 1
                end

end −− BUFFER
```

6.2 Alternative model

The code using this model is very similar, using an identical buffer:

```
class  ROOT_CLASS creation
        make
feature

        b:  BUFFER[INTEGER]
        p:  PRODUCER
        c:  CONSUMER

        make is
                do
                        create  b.make
                        async create  p.make(b)
                        async create  c.make(b)
                end
end −− ROOT_CLASS

class  PRODUCER creation
        make
feature
        buffer:  BUFFER[INTEGER]

        make (b: unlocked BUFFER[INTEGER]) is do buffer := b; keep_producing end

        keep_producing is
```

```
        do
                ...
                buffer.put(i)
                ...
        end

end -- PRODUCER

class CONSUMER creation
        make
feature
        buffer: BUFFER[INTEGER]

        make (b: unlocked BUFFER[INTEGER]) is do buffer := b; keep_consuming end

        keep_consuming is
        local
                ...
        do
                ...
                i := consume_from_buffer(buffer)
                ...
        end

        consume_from_buffer (b: BUFFER[INTEGER]) : INTEGER is
                require
                        not b.is_empty
                do
                        Result := b.item
                        b.remove
                ensure
                        b.count = old b.count - 1
                end

end -- CONSUMER
```

In such a small program, the differences arising between the original SCOOP model, and the alternative model, are few; in this case they are:

- **separate** is removed, since all the objects are notionally concurrent.
- In ROOT_CLASS.make, the creation of the buffer is unadorned, but the creation of the producer and consumer objects are annotated with **async** indicating that the calls to their creation procedures should be asynchronous calls.
- The argument (the buffer) to each of the creation procedures for the producer and consumer is annotated **unlocked**. The SCOOP version serialises these two creations due to the implicit reservation on the buffer.
- store_in_buffer is not required in the alternative version, since it merely exists to obtain a lock on the buffer and call put on the buffer. Thus a lazy lock suffices here.

– A lazy lock is not sufficient to replace consume_from_buffer

This example is usually generalised to the case of producing and consuming several items at once. In this case, more explicit mutual exclusion is required and a multi-item store_in_buffer would be required.

7 Work-in-progress: a CSP sketch of the alternative mechanism

Using the process algebra CSP [9], we sketch a model of our alternative formulation of concurrency for Eiffel in a style similar to that in the SCOOP model [2]. Briefly,

– $Stop$ denotes the process that does nothing;
– $a \rightarrow P$ performs event a then behaves as process P;
– $P \Box Q$ allows the environment to choose between processes P and Q via their first events. If the first events of P and Q are identical, then the choice of P and Q is nondeterministic.
– $Skip$ is the process representing successful completion.
– $P; Q$ behaves as P; then after P is successful, then behave as Q.
– $P|||Q$ interleaves the two processes P and Q.
– $P \|_A Q$ denotes the two processes P and Q agreeing on events in the set A and interleaving other events.
– $g \Rightarrow P$ is the process P if g is true, and is $Stop$ otherwise. (Some authors use $g \& P$ for this.)

We refer readers to Hoare's text (or a later text by Roscoe [13] or Schneider [14]) for more information about CSP.

As in the CSP model of SCOOP [2], we introduce the system in layers: objects, partitions and (finally) a system. Objects can be reserved; each can carry out work and make calls. All these aspects are represented as CSP programs.

We use several definitions similarly to the CSP model of SCOOP. A *system* comprises a number of objects, each of which has its own notional thread of control. The overall system comprises one or more *partitions* or processing resources, which may be physical CPUs, POSIX processes, individual threads, or any other processing model. Each partition is responsible for zero or more objects. Here, we are borrowing some elements and terms of the Ada 95 model of distributed processing [1].

7.1 System

A system is modelled by the parallel composition of its component partitions:

$$SYSTEM \triangleq \|_{k:partitions} PARTITION(k) \quad (1)$$

where $partitions$ is a set giving the names of all partitions in the system.

7.2 Partitions

Objects are grouped together and assigned to partitions. For each partition k, the intended semantics is that each object resident on that partition can progress independently (although a real implementation would have to share the processing resource). In CSP terms, we write

$$PARTITION(k) \triangleq \|_{j \in residents(k)} OBJECT(j) \tag{2}$$

where

$$residents(k) \triangleq \{j \text{ s.t. object } j \text{ is resident on partition } k\ \} \tag{3}$$

7.3 Reservations

We reason about objects being reserved by calls made by executing routines on a different object. In this sketch, each object is referred to simply by its name, say, j. Similarly, each call c embodies all the information about that call, such as the calling and called objects, arguments, and any other necessary bookkeeping information. We assume that calls are unique in the model (a recursive routine should have different calls associated with each instance of the recursive routine).

We define the function $co(c)$ ('caller object') such that it returns the object on which the call executes.

Our model requires information about 'call chains', e.g., the information that call c_1 called c_2 which itself called c_3. We represent this with the function $isCaller(c_1, c_2)$, which is true if and only if c_1 called c_2 directly (a possible variation might allow intermediate calls).

Our model also requires information about reservations on an object j. We associate a sequence of calls, R_j, with an object j:

- A call c is in R_j if c has a reservation on j, even if it has handed-on the reservation.
- The last call in R_j is the *active reservation*: all calls with earlier reservations have handed-on the reservation to a subsequent call.

So a call c has exclusive access to object j if and only if c is the last index in sequence R_j. Note that we allow c to remove itself from the list of reservations at any time: this represents the call indicating that it no longer has any interest in j (e.g., it has completed).

We say that an object j is *available for reservation* if either

- the object is totally unreserved, i.e., $R_j = \langle\rangle$; or
- the object is reserved by the (or a) caller higher up its call chain (modelled by $isCaller$).

We can now write down a process representing object j's reservation behaviour:

$$\begin{aligned} OBJECT_R(j, R_j) \triangleq\ & R_j = \langle\rangle \vee isCaller(last(R_j), c) \Rightarrow reserve.j.c \rightarrow OBJECT_R(j, R_j \frown \langle c \rangle) \\ & \Box R_j \neq \langle\rangle \wedge \neg isCaller(last(R_j), c) \Rightarrow blocked.j.c \rightarrow OBJECT_R(j, R_j) \\ & \Box free.j?c : \sigma(R_j) \rightarrow OBJECT_R(j, R_j \downarrow \{c\}) \\ & \Box unreserved.j?c : CALLS \setminus \sigma(R_j) \rightarrow OBJECT_R(j, R_j) \\ & \Box dying.j \rightarrow \mu X \bullet dead.j \rightarrow X \end{aligned} \tag{4}$$

where $s \downarrow A$ means the sequence s with all occurrences of members of the set A removed, and $\sigma(s)$ returns the set of elements contained in the sequence s.

Taking the clauses in the equation above one-by-one, they say:

- Call c can reserve j if j is totally unreserved, i.e., $R_j = \langle\rangle$, or if the active reservation on j was made by the caller of c.
- If c cannot reserve j, then the model only offers the $blocked$ event.
- c can free j (for itself) at any time, provided that c is currently reserving j.
- The next clause handles c attempting to free j when it did not have a reservation. (We need this for the CSP treatment of $RELEASING$, which we define later.)
- The last clause deals with the case where an object has died due to an unhandled asynchronous exception. It will never work again.

The mechanical model in the SCOOP paper [2] has several additional clauses that allow the transfer of state information between different parts of the model by engaging in events. We omit such aspects in this sketch.

7.4 Objects

Before an object can be reserved or carry out any work, it must be created.

Creation is effected via the Eiffel **create** keyword (or semantically equivalent methods). We represent this period by the two events $startCreation.j$ and $endCreation.j$, and the process

$$CREATION(j) \triangleq startCreation.j \rightarrow endCreation.j \rightarrow Skip \tag{5}$$

Calls are queued on objects Calls to other objects do not immediately execute: they are queued instead and executed in FIFO order. Each object is associated with a queue (sequence) of calls. The program representing the queue has the object's 'name', j, and a queue of calls, q, as parameters.

$$\begin{aligned} OBJECT_Q(j, q) \triangleq\ & last(R_j) = c \Rightarrow add.c?c' \rightarrow OBJECT_Q(j, q \frown \langle c' \rangle) \\ & \Box q = \langle c \rangle \frown q' \Rightarrow schedule.j!c \rightarrow OBJECT_Q(j, q') \\ & \Box dying.j \rightarrow \mu X \bullet dead.j \rightarrow X \end{aligned} \tag{6}$$

This program says that a call c that has the active reservation on an object j may enqueue calls c' on that object; that a call c at the front of the queue may be scheduled for work; or that an object might die.

Objects scheduling work The next program, $OBJECT_P$, performs work for one call at a time on an object.

$$OBJECT_P(j) \triangleq schedule.j?c \rightarrow DOCALL(c); OBJECT_P(j) \tag{7}$$

This program cooperates with $OBJECT_Q$ in $schedule$ events, then becomes $DOCALL$ from equation 17 in Section 7.5.

Objects completed An object j is first created (equation 5) and can then be reserved (equation 4) while carrying out work (using the $_Q$ and $_P$ programs):

$$\begin{aligned} OBJECT(j) \triangleq\ & CREATION(j); \\ & (OBJECT_R(j, \langle\rangle) \| OBJECT_Q(j, \langle\rangle) \| OBJECT_P(j)) \end{aligned} \tag{8}$$

7.5 Calls

We now concentrate on calls between objects.

Reserving call arguments A call entails a substantial amount of information: an object j_1 calling a routine r in j_2 where r takes one or more arguments, where any number of arguments may need to be reserved. We denote the *required arguments* as $\langle s_1, s_2, \ldots, s_p \rangle$: these are the objects referenced in the argument list, except for those annotated by **unlocked**.

From our model in Section 5, we see that

- j_1 must hold the active reservation on j_2 before it can call r in j_2, and
- j_2 needs to obtain reservations on each required argument before it can execute r.

In the following, the current instance of the execution of r on j_2 is labelled as the call c. We write $\mathbf{s^c}$ for the required arguments of call c, where $\mathbf{s^c} = \langle s_1^c, s_2^c, \ldots, s_{p_c}^c \rangle$.

Collecting reservations for a call We construct a process that collects the reservations (or notes the block) for each required argument:

$$\begin{aligned} RESERVING(c) \triangleq |||_{q:\{1,\ldots,p_c\}} \big(& reserve.s_q^c.c \rightarrow Skip \\ & \Box\, blocked.s_q^c.c \rightarrow Skip \big) \end{aligned} \tag{9}$$

Note that we allow the reservations to proceed in any order.

All reservations have been obtained for call c if for all s_q^c in $\mathbf{s^c}$

$$last(R_{s_q^c}) = c$$

A CSP process CR (short for '$CHECKRESERVATIONS$') walks through $\mathbf{s^c}$, and if for each s_q^c can engage in $gotLock.s_q^c.c$, then all reservations were successfully made:

$$CR(c) \triangleq CR'(c, \mathbf{s^c}) \tag{10}$$

$$\begin{aligned} CR'(c, \mathbf{s}) \triangleq\ & \mathbf{s} = \langle\rangle \Rightarrow Skip \\ & \Box\ \mathbf{s} = \langle s_q^c \rangle^\frown \langle \mathbf{s'} \rangle \Rightarrow gotLock.s_q^c.c \rightarrow CR'(c, \langle \mathbf{s'} \rangle) \\ & \Box\ \mathbf{s} = \langle s_q^c \rangle^\frown \langle \mathbf{s'} \rangle \Rightarrow notGotLock.s_q^c.c \rightarrow SUSPEND(c, \mathbf{s^c}) \end{aligned} \tag{11}$$

Suspending a call If all the reservations required cannot be made, then the call releases all the reservations it did manage, then suspends to reattempt the call later.

A simple CSP program represents suspension of a call c:

$$\begin{aligned} SUSPEND(c) \triangleq\ & RELEASING(c); \\ & suspendCall.c \rightarrow reattemptCall.c \\ & \rightarrow DOCALL(c) \end{aligned} \tag{12}$$

where $RELEASING$ is described immediately below; $DOCALL$ is described in Section 7.5; and the events $suspendCall.c$ and $reattemptCall.c$ simply mark the passing of time between themselves.

Releasing reservations We construct a counterpart process to equation 9 that frees all reservations collected:

$$\begin{aligned} RELEASING(c) \triangleq\ & |||_{q:\{1,\ldots,p_c\}} \big(free.s_q^c.c \rightarrow Skip \\ & \Box unreserved.s_q^c.c \rightarrow Skip \big) \end{aligned} \tag{13}$$

We need do nothing more when releasing reservations: either we have the reservation, in which case we can release it, or we never had it, so we skip over it (the $unreserved$ event).

Wait conditions of a call A call does much more than make and release reservations. Once it has collected all its reservations (which might have taken several attempts) it checks the wait conditions on the call. If the wait conditions are false, then the call suspends (i.e., releases all its reservations and tries again later).

Since we do not model the detailed semantics of sequential Eiffel code, we instead offer several events:

- $checkStart.c$: the start of the evaluation of wait conditions and preconditions.

- $failWait.c$: the wait conditions are not true (so suspend).
- $failWaitAlways.c$: the wait conditions are not true; moreover, they will never be true.
- $proceed.c$: the wait conditions are true, so the call proceeds.

The following CSP program represents the process of checking these conditions.

$$\begin{aligned} CHECKCONDITIONS(c) \triangleq\ & checkStart.c \rightarrow \\ & \Big(proceed.c \rightarrow Skip \\ & \Box failWait.c \rightarrow SUSPEND(c) \\ & \Box failWaitAlways.c \\ & \qquad \rightarrow dying.co(c) \rightarrow Stop \Big) \end{aligned} \tag{14}$$

Calls doing work Once all reservations have been made and the wait conditions are true, then the work of the call can go ahead, represented by the CSP program $DOWORK(c)$:

$$DOWORK(c) \triangleq startWork.c \rightarrow WORK(c); endWork.c \rightarrow Skip \tag{15}$$

where the events $startWork$ and $endWork$ represent time passing during the work. The program $WORK(c)$ is the representation of call c's actual work.

Calls making calls As part of $WORK(c)$, the call may make other calls: call c may attempt call c'. There are two cases: the first is that the object is calling itself — in this case, it immediately becomes $DOCALL(c')$.

The second case is an inter-object call. This is represented by the single event $add.c.c'$. $add.c.c'$ can only occur if c has the active reservation on c'. This is enforced in the object model above in equation 6.

If c requires the result of c', then it synchronises on the event $endWork.c'$.

We represent this by the CSP program $ADD(c, c')$:

$$\begin{aligned} ADD(c, c') \triangleq\ & add.c.c' \rightarrow \\ & \big((isSynchronous(c') \Rightarrow endWork.c' \rightarrow Skip) \\ & \Box (\neg isSynchronous(c') \Rightarrow Skip) \big) \end{aligned} \tag{16}$$

The function in the guards, $isSynchronous(c')$, is a constant boolean function that tells the model whether call c needs to synchronise on the end of call c'.

Calls making lazy locks The alternative model in this paper allows for lazy locks. This manifests in the CSP sketch by additional synchronisations on $reserve$ and $free$ events. For instance, if the body of c requires a lazy lock on object j to make call c', then the CSP

$$reserve.j.c \rightarrow ADD(c, c'); free.j.c \rightarrow Skip$$

models the lazy reservation of j, the enqueue of call c', and the release of j immediately afterwards. Note that this causes the call c to block until it can reserve j.

Unhandled asynchronous exceptions During the execution of a call c (or during the evaluation of preconditions, wait conditions, etc.), an exception may be raised. If this causes c to fail, then this exception is an unhandled asynchronous exception as in point 9 of Section 5. Moreover, because we do not model sequential calls, the call c causes its owning object to die.

This is represented in our CSP sketch by the offending part engaging in the event $dying.co(c)$. The event $dying.j$ is engaged in by processes $OBJECT_R(j)$ and $OBJECT_Q(j)$, with the result that the object only ever engages in $dead.j$ events thereafter (where $j = co(c)$).

This sketch does not directly represent the subsequent exceptions that are raised in future callers.

Putting it together: executing a separate call A call c is scheduled to be executed. We can collect together all the programs above thus:

$$\begin{aligned} DOCALL(c) \triangleq\ & RESERVING(c); \\ & CR(c); CHECKCONDITIONS(c); \\ & DOWORK(c); RELEASING(c) \end{aligned} \tag{17}$$

where $DOCALL$ is used in equation 7 (Section 7.4).

At this point, we have completed our model.

8 Discussion

8.1 Remarks on our model

We allow compilers to implement synchronous calls asynchronously if they can guarantee semantically equivalent behaviour. This is so that multiple processing nodes can be utilised in parallel. However, there is the question of defining 'semantically equivalent behaviour'.

Lazy locks may admit deadlocks when old sequential code is used as a component in concurrent programs, but this is better than allowing races. Without lazy locks, our model would result in most (if not all) existing sequential programs having missing locks in many places. Even then, some sequential code may not work in a concurrent environment (due to races by interleaving calls from multiple concurrent clients, or deadlocks by other clients locking a callee). We need to evaluate this further, perhaps by implementation in a transparent compiler (the subject of a proposal itself).

We initially considered the use of both preconditions *and* wait conditions. However, we have been unable to justify separate preconditions. Instead, this

model allows an immediate synchronous exception when it is clear that a particular wait condition will never be satisfied.

So far, we have not addressed real-time programming, although a utility class could offer 'delay' and 'delay until' (as for Ada [1]).

The CSP sketch is incomplete and also requires mechanical implementation to validate it. It is closely related to the model proposed for SCOOP [2], but is obviously simpler — although this model is undergoing continual revision. In particular, the sketch in this paper removes subsystem entirely and adds a very limited model of asynchronous exceptions. Much of the rest of the model is similar. Whether this translates into improved performance and clarity of understanding is a different matter, and one that we will address in the continuation of this work.

8.2 Comparison with SCOOP

We compare our model against SCOOP and the problems in Section 4.

1. *Chains of calls and reservations.* Although we identify objects to be locked differently, the handing-on of reservations through call chains is the same mechanism that we propose for SCOOP itself.
2. *Release of reservations.* Lazy locks are released as soon as the call is enqueued; other locks are only released at the end of the feature concerned.
3. *Reduced parallelism by subsystems and implicit reservations.* This model removes subsystems, but still requires reservations to enqueue feature calls. Whether this produces practical results is unclear.
4. *Rescheduling of blocked calls.* We have explicitly given some details of the handling of suspended calls, although we recognise that more details are needed. In particular, real-time behaviours would likely give a different set of requirements.
5. *Priorities and call queues.* Calls are enqueued in both models, although SCOOP enqueues them against the subsystem (potentially many objects) whereas our model is queued per-object. The objects themselves can be given priorities for processing by scarce resources, although there are a number of real-time programming issues still to address.
6. *Complications of separate-ness (subsystems, traitors, reservation via arguments).* This model does not contain subsystems. A mechanism is included to allow objects to be given in argument lists without implicitly reserving them.
   We claim that this makes this model simpler: we have discarded entirely the whole concept of handlers and subsystems. Because all objects are 'separate', we no longer require the concept of traitors, or separateness consistency rules. Thus, the type system no longer needs fixing for library calls (e.g., having separateness being a dimension of type).
   Making all objects concurrent gives a more symmetric semantics (we no longer find some objects local with respect to others). However, our model does not break existing sequential programs because asynchronous calls are made explicitly rather than implicitly (via a call on a separate object) as in SCOOP.

7. *Omission of asynchronous exceptions, real-time and interrupts.* Mechanisms are given for handling asynchronous exceptions and interrupts. We propose both mechanisms for inclusion in SCOOP itself. We have not attempted to address real-time.
8. *Implementation matters.* Most of these issues can be solved equally for SCOOP and our model, but we have not attempted to do so here. Both require a CCF (for mapping objects to real resources) and deadlock detection. The notion of partitions is given in this model, and an explicit exception identified for failure of the underlying implementation.
   Notably, scalability remains an open question for this model.

As well as the points above, we should also consider the pragmatics of programming using the notation. This needs to be tested with large programs.

9 Conclusion

We have briefly described SCOOP, a major existing proposal for adding concurrency to Eiffel. After identifying some problems with SCOOP, we have proposed an alternative model and sketched a model in CSP.

The next steps in our work are

1. Add further details to the CSP sketch.
2. Implement the CSP model mechanically, and analyse for undesirable behaviours.
3. Implement this either as a preprocessor to an existing Eiffel compiler, or in a specially-produced compiler.
4. Assess requirements for, and implement any needed support for real-time programming, and safety- and security-critical systems.
5. Develop techniques for proof and model-checking.

Acknowledgements

We thank the referees, Jeremy Jacob, Jonathon Ostroff, Piotr Nienaltowski and the ETHZ SCOOP researchers for their helpful comments and discussions.

References

1. T. Taft and R. A. Duff, editors. *Ada 95 Reference Manual*. Number 1246 in Lectures Notes in Computer Science. Springer-Verlag, 1997.
2. Phillip J. Brooke, Richard F. Paige, and Jeremy L. Jacob. A CSP model of Eiffel's SCOOP. Submitted for publication, November 2005.
3. Phillip J. Brooke and Richard F. Paige. A critique of SCOOP. To appear in *Proc. CORDIE 2006*, University of York Technical Report, July 2006.
4. D. Caromel. Towards a method of object-oriented concurrent programming. *Comm. ACM* 36(9), September 1993.

5. M. Compton. *SCOOP: an Investigation of Concurrency in Eiffel*, MSc Thesis, Australian National University, 2000.
6. ECMA-367: Eiffel Analysis, Design and Programming Language, Ecma International, June 2005.
7. Formal Systems (Europe) Ltd. Failures-Divergence Refinement: FDR 2. http://www.formal.demon.co.uk/, December 1995.
8. O. Fuks, J.S. Ostroff, and R.F. Paige. SECG: The SCOOP-to-Eiffel Code Generator. *Journal of Object Technology* 3(10), November/December 2004.
9. C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.
10. B. Meyer, Object-Oriented Software Construction, 2nd Edition, Prentice Hall, 1997.
11. P. Nienaltowski, V. Arslan and B. Meyer, SCOOP: Concurrent programming made easy, draft paper, http://se.inf.ethz.ch/people/nienaltowski/papers/scoop_easy_draft.pdf, 2004.
12. P. Nienaltowski, SCOOPLI implementation, http://se.inf.ethz.ch/research/scoop.html, 2005.
13. A.W. Roscoe. *The Theory and Practice of Concurrency*. Series in Computer Science. Prentice Hall, 1998.
14. Steve Schneider. *Concurrent and Real-time Systems*. Wiley, 2000.