SIMPLE CONCURRENT OBJECT – ORIENTED PROGRAMMING:

A GENERATOR BASED IMPLEMENTATION

by

Oleksandr Fuks

A Thesis (will be) Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

MASTER OF SCIENCE

YORK UNIVERSITY

2 0 0 4

Copyright Page (ii — not typed)

Certificate Page (iii — not typed)

Abstract

Concurrent programming is notoriously error prone. SCOOP is a simple but powerful notation for concurrent programming built on top of standard Eiffel (Meyer 1997). The SCOOP extension to standard Eiffel covers fully-fledged concurrency and distribution constructs, but is as minimal as it can get. Starting from the standard sequential Eiffel notation, there is the addition of a single new keyword — **separate**. This simplifies mutual exclusion and synchronization, and almost completely removes problems such as the inheritance anomaly.

In this thesis, we describe a SCOOP to Eiffel *Generator.* The Generator is the first workable and complete cross-platform implementation of SCOOP. We show how SCOOP constructs can be mapped to standard Eiffel and the use of a cross-platform threads library (Eiffel+Threads). The Generator automatically converts SCOOP programs to running Eiffel+Threads code.

Eiffel has powerful features such as Design by Contract, genericity, multiple inheritance, and seamless and reversible design and code generation via BON. The addition of a SCOOP concurrent facility, fully compatible with all the standard Eiffel features, makes the resulting framework a productive environment for developing quality concurrent code.

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Jonathan Ostroff, whose expertise, understanding, and patience added considerably to my graduate experience. I appreciate his vast knowledge and skill in many areas much beyond the limits of Computer Science. His personal qualities and his life experience never stopped surprising me, and always made me think of him as of a person who is a great example of modesty, wisdom and kindness. I cannot overestimate Jonathan's contribution to this project.

I also would like to thank Dr. Vassilos Tzerpos for suggestions on my thesis and for serving as a member of my thesis committee.

I would like to thank the other members of my committee, Dr. Yves Lesperance, and Dr. Luiz Marcio Cysneiros for the assistance they provided at all levels of the research project.

My special thanks goes to my friend, Borys Derevyanchenko, who was always there when I needed any fresh idea or just a push. My special thanks also to David Makalsky for his many good suggestions and editorial changes.

I would like to thank my close friend Ian Wahn who's editing assistance and an ability to always be there for me helped me a lot while writing this thesis.

I would also like to thank my family for the support they provided me through my entire life and in particular. They are as happy for my every achievement as I am.

Table of Contents

List of Figures

Chapter 1 – Introduction

Concurrent programming is considered to be a challenging and inherently error prone process. The continuing discussions of flaws in the Java memory model (originally described in Chapter 17 of the Java Language Specification) is a symptom of the difficulties. The Java memory model gives constraints on how threads interact through memory. But the model was hard to interpret and poorly understood. Many JVMs actually violated the constraints of the memory model (Lea 1999), and thus there has been a concerted and ongoing effort to eliminate the flaws.

In this thesis we provide an implementation of SCOOP (Simple Concurrent Object Oriented Programming) as defined by Meyer (Meyer 1997) for concurrent programming. SCOOP provides a simple framework for concurrent development that also helps the developer to isolate and avoid common problems. The nice integration of Object-Oriented Design, contracts and the simple concurrency model of SCOOP is a good motivation for developing actual executable target code. We describe and implement a prototype *Generator*. As part of the work for this thesis:

- a manual, the code and the Generator executable was first made available in March 2003 to the Eiffel and open source communities at the URL *http://scoop2eiffel.sourceforge.net*; and
- a journal paper describing SCOOPGEN is to appear in the November/December 2004 issue of *JOT - Journal of Object Technology.*

Many mechanisms exist for introducing concurrency into object-oriented (OO) programming languages. The pervasiveness of multi-tasking operating systems, in which several programs can use various resources concurrently, has increased the potential of parallel computations. These approaches support the use of multiple, and sometimes distributed processors, each of which may be executing multiple processes. Different techniques are provided with various languages to support synchronization, interruption, mutually exclusive access to object state, and atomic execution of routines.

However, the process of development and, in particular, debugging of programs using parallel programming is complex and labour-intensive resulting in large financial expenses due to programmer time (McDowell 1989). To implement concurrency, compiler writers had to use special hardware/system calls. To bring concurrency up into the programming language and out of low-level system calls, language developers started adding concurrency language constructs into the language and compiler, to support automatic translation into the appropriate low-level behaviour. Use of these language constructs allows developers to treat concurrency at an abstract level, not wasting time and effort on the details of the implementation of parallel calculations.

In the late 1980s, there was a paradigm shift in programming, as Object-Oriented languages became prevalent. With popular Object-Oriented languages such as Modula-3, SmallTalk, Eiffel and C++, development time was reduced, program analysis was simplified and code reuse was made possible via information hiding and encapsulation. Subsequently, language constructs were also added to implement parallel calculations in Object-Oriented languages (Tsichritzis 1995).

There are various approaches to concurrency in object-oriented programming languages. The development of concurrency constructs is found in languages such as C++, Java, SmallTalk (which uses Active Objects) and Eiffel.

In C++, two approaches have been used to add concurrency. In the first approach, the language is extended in order to add the concurrency constructs. The second approach uses the facilities of OOP to encapsulate the lower-level details of concurrency in a library. In the library approach, a library class (generally referred to as a Task class) provides the concurrent facilities. A user wishing to write concurrent code can use Task, normally by inheriting from it. In this library approach, the concurrency constructs are kept outside of the language. As stated in (Arjormandi 1995), the library approach "keeps the language small, allows the programmer to work with familiar compilers and tools, provides the option of supporting many concurrent models through a variety of libraries, and eases porting of code to other architectures (usually, a small amount of assembler code needs to be changed). Software developers typically have large investments in existing code and are reluctant to adopt a new language. A class library with sufficient flexibility that can provide most of the functionality of a new or extended language is often more palatable. On the other hand, new or extended languages can use the compiler to provide higher-level constructs, compile-time type checking, and enhanced performance".

Concurrency is currently supported in Eiffel via the library approach. However, Meyer (Meyer 1997) has provided an approach called SCOOP (see below) for extending the language with concurrency. The novelty of Meyer's approach is that only one new keyword "**separate**" is required. Yet this single construct provides all the main properties of concurrent computation, even simplifying the resultant code.

(Compton 2000) was the first to implement SCOOP. In Compton's work, SCOOP is implemented via changes in the open source SmallEiffel compiler. However, this implementation of SCOOP is now incompatible with later versions of the SmallEiffel compiler (now called SmartEiffel). It also did not implement the full set of SCOOP constructs (such as contracts and "once" routines). A *once* routine has a body that will be executed only once, for the first call; subsequent calls will have no further effect and, in the case of a function, will return the same result as the first. This provides a simple way of sharing objects in an object-oriented context.

In this thesis, we provide the first full implementation of SCOOP in a multi-threaded setting[1] that is fully compatible with the current commercial Eiffel Software compiler (www.eiffel.com). This work is reported (in part) in the journal article (JOT 2004).

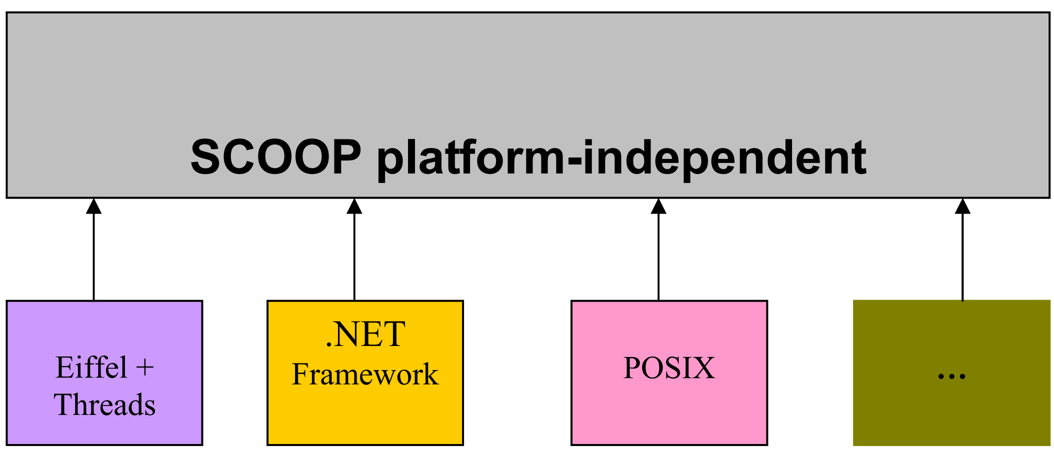

Figure 1-1 SCOOP Architecture

SCOOP has a two-level architecture as shown in Figure 1-1. The top layer is the platform independent layer. A SCOOP program (at the top layer) can be implemented on different underlying platforms (such as Posix and .NET) as shown in the bottom layer in

[1] This implementation does not support distributed computation via CCF files and duel mechanism interrupts.

the figure. The implementation in this thesis is in terms of a multi-threaded model (see the box labeled "Eiffel+Threads").

Subsequent to the work reported in this thesis, another implementation of SCOOP in the .NET framework has been developed (Nienaltowski 2003). We will compare our approach and the .NET approach in the sequel, but currently, the .NET implementation is not as complete as our approach (e.g. it does not support exclusive locking of multiple concurrent objects). On the other hand, the .NET implementation allows for distributed computing.

1.1 Eiffel and SCOOP

Eiffel includes many modern object-oriented language features through which it aids developers in creating robust, reusable, secure, extensible, portable and maintainable software (Meyer 1997). Eiffel supports Design by Contract (DbC), genericity, multiple inheritance, static typing/dynamic binding, garbage collection, "once" routines, self-documentation, and other advanced language features.

As mentioned earlier, the Eiffel language can be provided with concurrency constructs via SCOOP (Simple Concurrent Object-Oriented Programming). The concurrency constructs of SCOOP extend the Eiffel language by adding one keyword ("`separate`") that can be applied to classes, attributes, and formal routine arguments. The application of `separate` to a class (or equivalently, declaring an attribute associated with a class as `separate`) indicates that the class executes in its own thread of control. The application of `separate` to routine arguments indicates that these objects are points of synchronization, and can be safely shared among concurrent threads.

The commercial Eiffel Software compiler, as well as the open source SmartEiffel compiler, are both planning to implement SCOOP. The Eiffel Software compiler already reserves the `separate` keyword to this end, although no implementation of SCOOP has been released yet (ISE 2003).

In this thesis we will describe a tool, called the *SCOOPGEN Generator*. The Generator translates Eiffel SCOOP programs (using the `separate` keyword) into standard Eiffel threaded applications (that make use of Eiffel's `THREAD` class). This approach has multiple benefits:

- The resulting code is pure Eiffel that compiles on standard Eiffel compilers (provided the compiler supports Eiffel Software's `THREAD` class).
- Class `THREAD` is described in detail in Appendix A, and its implementation is in terms of standard POSIX threads. It is relatively easy to port it to other compilers such as SmartEiffel.
- The Generator is not dependent on changes to the standard Eiffel compilers. Only significant changes to Eiffel syntax would require (probably minor) changes to the Generator.
- The target code will run on any platform supported by the compiler. For example, Eiffel Software's compiler runs on Windows, Linux, Macintosh and various embedded systems.

The main disadvantage of this approach is that debugging must currently be performed in the standard runtime systems of the target code rather than being able to work at the abstract level of SCOOP code.

The Generator is implemented and works successfully with the latest Eiffel Software compiler and Integrated Development Environment *EiffelStudio* (Version 5.4).

As mentioned earlier, (Compton 2000) was the first to implement SCOOP. In Compton's work, SCOOP is implemented via changes in the open source SmallEiffel compiler, and its runtime system and debugger thus has the advantage of supporting SCOOP programs directly. However, this implementation of SCOOP is now incompatible with later versions of SmallEiffel compiler (now called SmartEiffel). It also does not implement the full set of SCOOP constructs (such as contracts and "once" routines).

. The *producer-consumer* example in the next subsection will illustrate some of the features of a SCOOP program.

1.2 A Producer-Consumer example

The producer-consumer problem illustrates the need for synchronization in systems where many processes share a resource. In this section, we will provide an informal introduction to SCOOP using this problem.

In the producer-consumer problem, two processes share a fixed-size buffer. One process produces information and puts it in the buffer, while the other process consumes information from the buffer. These processes do not take turns accessing the buffer, they both work concurrently. Herein lays the problem. What happens if the producer tries to put an item into a full buffer? What happens if the consumer tries to take an item from an empty buffer? In order to safely synchronize these processes, we (a) use some mechanism to provide mutual exclusion so that only one process at a time can access the buffer (otherwise the information in the buffer might be garbled), and (b) we must block the producer when the buffer is full, and block the consumer when the buffer is empty.

A standard Java solution is shown in Appendix D (Listing 6). Three separate constructs are needed for the final solution. (a) Class PRODUCER and CONSUMER must inherit from a **THREAD** class, (b) the *put* and *get* methods of BUFFER must be declared **synchronized**, and (c) the *put* and *get* methods must **wait()** to be notified (via **notifyAll()**)that the buffer is available. Alternatively, we may use a `sleep` method instead of `wait/notify` (to ensure that we do not use up CPU cycles with an unnecessary busy-wait).

The SCOOP version of the producer-consumer provides the same behavior as the Java solution, but with the simplification that only one extra keyword `separate` is used (instead of `Thread`, `synchronize` and `wait/notify`). In addition, the SCOOP solution uses contracts with all the benefits of DbC, although as we will see, the meaning of a precondition will change (postconditions, class invariants, and loop variants and invariants retain the original semantics).

The BON diagram shown in Figure 1-2 specifies the various classes. The ROOT_CLASS (shown in Figure 1-6) has three attributes: buffer `b`, producer `p` and consumer `c`. The buffer `b` (of type `BUFFER`) is declared `separate,` thus indicating that it executes in its own logical thread (called a *subsystem*). This means that `BUFFER` (Figure 1-7) is just a standard class having routines `put` and `remove` (without any regard to concurrency). Thus it has no concurrent keywords in it, and when used in sequential programs has none of the concurrent overheads. By declaring buffer attribute `b` in the root class `separate`, we thereby specify that it executes in its own subsystem and that all its routines are "synchronized" (using Java notions).

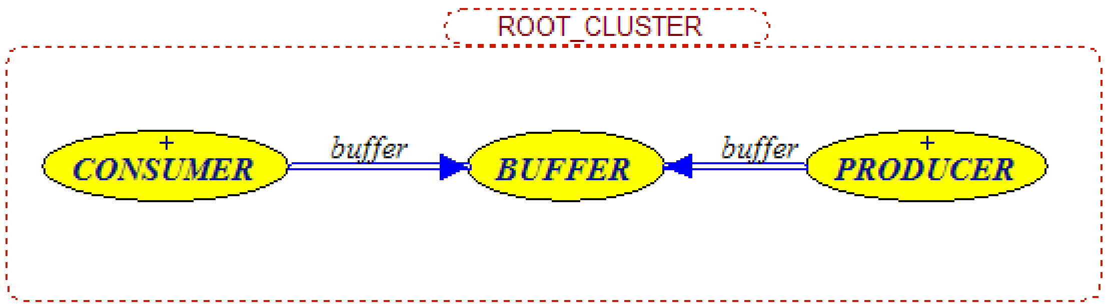

Figure 1-2 'producer-consumer' BON Diagram

The contracts of `BUFFER` are as expected (Figure 1-7). For example, the routine `put` in `BUFFER` has a precondition that asserts that you cannot put more than three elements in the buffer. Its postcondition asserts that after a `put`, the number of items in the buffer is incremented by one, and that the buffer actually has the new element inserted.

By contrast to BUFFER, classes PRODUCER and CONSUMER are declared `separate` right at the beginning (Figure 1-3 and Figure 1-5). This is because they are inherently concurrent and always execute in their own subsystems. When classes PRODUCER and CONSUMER are first created (via their constructor routine `make`), we pass to them (as an argument of `make`) the reference to the same buffer (`b`). Thus both PRODUCER and CONSUMER have an attribute

```
buffer: separate BUFFER
```

to store this reference to `b`. Attribute `buffer` must be declared `separate` to indicate that its routines execute in a different subsystem to the current object (either the producer or the consumer).

```
separate class PRODUCER create
      make

feature  {NONE}
      buffer: separate BUFFER

      make (b: separate BUFFER) is
                  -- Initialize `Current'.
            do
                  buffer := b
                  keep_producing
            end

      keep_producing is
            local
                  i: INTEGER
            do
                  from
                  until
                        False
                  loop
                        i := (i + 1) \\ 5
                        produce (buffer, i)

                              -- buffer.put or buffer.remove

                              -- is forbidden here
                  end
            end
      produce (b: separate BUFFER; i: INTEGER) is
            require
                  b.count <= 2
            do
                  b.put (i)
            end
end  -- class PRODUCER
```

Figure 1-3 PRODUCER class for ‘producer-consumer’ example

PRODUCER (via routine `produce`) and CONSUMER (via routine `consume`) must not access the buffer at the same time. Normally, we would protect the buffer with a *mutex* or a similar construct.

```
            produce (b: separate BUFFER; i: INTEGER) is
                  require
                        b.count <= 2

                        i >= 0
                  do
                        b.put (i)
                  end
```

Figure 1-4 produce routine for ‘producer-consumer’ example

```
separate class  CONSUMER create
      make
feature  {NONE}

      buffer: separate  BUFFER

      make (b: separate BUFFER) is
                  -- Initialize `Current'.
            do
                  buffer := b
                  keep_consuming
            end


      keep_consuming is
            do
                  from
                  until
                        False
                  loop
                        consume (buffer)
                        -- buffer.put or buffer.remove is forbidden here
                  end
            end

      consume (b: separate  BUFFER) is
            require
                  b.count > 0
            do
                  b.remove
            end

end  -- class CONSUMER
```

Figure 1-5 CONSUMER class for ‘producer-consumer’ example

```
class
      ROOT_CLASS
 create
      make
 feature  -- Initialization

      b: separate BUFFER
      p: PRODUCER
      c: CONSUMER

      make is
                  -- Creation procedure.
            do
                  create  b.make
                  create  p.make (b)
                  create  c.make (b)
            end
end  -- class ROOT_CLASS
```

Figure 1-6 ROOT_CLASS for ‘producer-consumer’ example

In SCOOP, we use argument passing, where the argument of a routine is declared `separate`, as a reservation (or synchronization) mechanism. For example, routine `produce` is presented on figure 1-4. A call to this routine will block until (a) the producer gets sole access to the buffer, and at the same time (b) the buffer must not be full as indicated in the precondition. If either (a) or (b) is false, the call waits until both are satisfied. Thus both mutual exclusion and the validity of the contract are ensured. A precondition clause involving a call with a `separate` target (`b.count <= 2`) is called a *separate precondition*. The other clause (`i >= 0`) is not `separate`.

However, the meaning of the precondition has now been changed. In sequential processing, the precondition is a correctness condition. If the precondition is true execution immediately proceeds to the body of the routine. If the precondition is false, an exception is generated. In the concurrent case, the precondition becomes a wait condition and the producer waits until the precondition evaluates to true.

```
class BUFFER create
      make

feature

      count: INTEGER is
            do
                  Result := q.count
            end

      item: INTEGER is
                  -- front
            do
                  Result := q.item
            end

      put (x: INTEGER) is
                  -- enqueue `x'
            require
                  count <= 3
            do
                  q.put (x)
                  io.new_line
            ensure
                  count = old count + 1
                  q.has (x)
            end

      remove is
                  -- dequeue
            require
                  count > 0
            do
                  q.remove
                  io.new_line
            ensure
                  count = old  count - 1
            end

feature  {NONE}

      q: QUEUE [INTEGER]

      make is
                  -- initialize buffer
            do
                  create  {ARRAYED_QUEUE [INTEGER]} q.make (3)
            end

 invariant
      0 <= count and count <= 3
end  -- class BUFFER
```

Figure 1-7 BUFFER for ‘producer-consumer’ example

It is only a `separate` precondition that delays. A non-separate precondition will act as a regular correctness condition.

How would we implement a SCOOP program into executable target code using POSIX-like threads and mutex locks? To call `consume` from routine `keep_consuming`, the consumer will pass `buffer` as an argument. When one or more arguments of a routine are `separate` objects, the client must obtain exclusive locks on all these objects before executing the routine. In our case, the consumer object must obtain an exclusive lock on `buffer` before executing `consume`. If another object (e.g. the `producer`) is currently holding the lock, the client must wait until the lock has been released, then try to acquire it. A default policy of first-in/first-out can be adopted. As described in (Meyer 2003) when the client succeeds in acquiring the lock:

- The `separate` precondition clauses are evaluated. If they all hold, the routine will execute, and then release the lock.

- Otherwise, the object releases the lock and restarts the whole process from the beginning: acquiring the locks, and then checking the `separate` precondition clauses. This allows other clients to access the supplier object and change its state, so that the wait conditions required by our client may eventually be met.

The locking policy facilitates building correct concurrent programs and reasoning about them:

- No interference between client objects is possible since at most one client may hold a lock on a supplier object at any time. This helps find which object is responsible for possible breaches in the contract, such as breaking the supplier invariant.

- The precondition rules ensure that correct calls do not violate the integrity of the supplier object.

1.2.1 `separate` call rule

As shown in Figure 1-3, we make it a syntactic error to call `buffer.put` in the routine `keep_producing` of the producer. This is because `buffer` is declared as a `separate` supplier. Instead we wrap the call in `produce` as discussed in the previous section. The main advantage of this approach is that the programmer does not need to worry about how to get access to the target object: this was taken care of by the call to produce, which had to reserve the object waiting if necessary until it is free.

SCOOP makes this scheme the only one for `separate` calls (i.e. calls to separate objects' routines) by introducing the *Separate Call Rule*, which asserts that the target of a `separate` call must be a formal argument of the routine in which the call appears. This rule may appear to put an undue burden on the developer of concurrent programs. In fact, what it really does is encourage developers to identify accesses to `separate` objects and separate them from the rest of the computation. This will actually help the developer avoid common concurrent development errors that normally make concurrent programming an error prone undertaking. We provide two examples to illustrate how SCOOP promotes good concurrent programming while helping the developer to avoid problems.

As one illustration of reservation via separate arguments, suppose we want to remove two integers, one after the other, from the buffer. The normal code

```
buffer.remove;
buffer.remove
```

will not work because any other client might jump in and interrupt (and hence disrupt) the execution. What we must do is wrap the above code in a routine with a `separate` argument:

```
remove_two (b: separate BUFFER) is
        do
              b.remove;
              b.remove;
        end
```

We can do the double remove merely by invoking the call `remove_two (buffer)`.

As another example, consider the code

```
if not buffer.empty then
      value := buffer.item
      buffer.remove
end
```

Without protection on `buffer`, another client may add or remove an element between the calls to `item` and `remove`. What makes things really bad is that the runtime behaviour is non-deterministic since it depends on the relative speeds of the clients. The bug will be intermittent and hard to reproduce. By encapsulating this error prone code in a `separate` routine, all these problems are eliminated.

1.2.2 Wait by necessity

A `separate` call to a supplier object only blocks until it acquires the resource and checks the preconditions as described above. The `separate` routine then executes its body in its own subsystem, and the calling object continues with the next statement in its own subsystem, i.e. it can continue with the rest of its computation.

Later on, the client may need to resynchronize with the supplier. Rather than introducing a specific language mechanism for this purpose, SCOOP relies on a "wait by necessity" mechanism in which the client waits on a *query* (but not on a *command* routine).

Consider the following code

```
1. x: separate X
…
2. x.compute_fourier_transform
3. do_some_other_processing
4. y := x.get_fourier_transform     -- wait by necessity
5. print(y)
```

In Java, as an example, execution would be blocked at line 2 until the routine to compute the Fourier transform runs to completion.

As explained above, *wait by necessity* just means that we do not block on commands, only on queries. As will be explained in more detail in 3-3, there is a refinement to *wait by necessity* introduced by (Compton 2000). However, in this thesis, we use the basic mechanism as explained above and as recommended by (Meyer 1997).

1.3 SCOOP syntax

The buffer example in the previous section illustrates the complete SCOOP syntax, i.e. we add to Eiffel the extra keyword `separate`. A `separate` SUPPLIER may be declared either as

- `x: separate SUPPLIER`, or
- `separate class SUPPLIER .. end`
  `x: S`

Suppose C1 is a `separate` class and C2 is an ordinary class. A `separate` routine call `r` in some class has the general form

```
r (x1: C1; x2: C1;
y1: separate C2; y2: separate C2; z: C2) is
    do
        x1.feature_1
        y1.feature_2
        z.feature_2 … -- etc.
    end
```

i.e. you may have as many arguments of any type as you want.

1.4 Contribution and organization of this thesis

Concurrent programming is an inherently difficult undertaking. We have argued that SCOOP as defined by Meyer (Meyer 97) provides a simple framework for concurrent programming that also helps the developer to isolate and avoid common problems. The nice integration of OO, contracts and the simple concurrency model of SCOOP is a good motivation for developing actual executable target code. Hence, the contribution of this thesis is to develop a SCOOP-to-Eiffel Code *Generator* that will

- parse SCOOP programs using the syntax outlined in Section 1.3;
- detect syntax errors in the SCOOP code such as violations of the `separate` call rule;
- translate syntactically correct SCOOP programs to standard Eiffel code that uses the Eiffel POSIX libraries for multi-threaded applications, so that the target code behaves according to the SCOOP semantics (outlined informally in Section 1.2).

The Generator is itself written in Eiffel.

The organization of this thesis is as follows:

- In chapter 2 we review existing approaches to concurrent OO programming.
- In chapter 3, we develop the SCOOP model in more detail than the original presentation in Meyer (Meyer 1997). The additional details were needed for implementation.
- In chapter 4, we describe the Generator in detail using the model developed in chapter 3.
- Chapter 5 provides the final discussion and conclusions.

Chapter 2 – Related Work

The idea of integrating concurrent or parallel computation into the object-oriented programming paradigm received wide acceptance relatively recently. There are many approaches to integration, as testified by extensive activity in this area.

The authors of (Briot 1998) define three basic approaches that make it possible to carry out the integration of parallel computation in object-oriented languages. These approaches include the *library* approach, the *integrative* approach, and the *reflective* approach. We discuss each of these approaches, and also their specific implementations. The SCOOP mechanism, implemented in this thesis, can be classified as integrative (using s*ynchronized objects*). Therefore attention in this chapter will be given mostly to a description of the integrative approach and method.

2.1 The Library Approach

In the library approach, class libraries are developed that make the implementation of parallel computation possible. These libraries include classes that encapsulate different components, necessary for parallel programs, such as threads, semaphores, critical sections, mutexes and others. This makes it possible to develop parallel programs (and thus to increase the effectiveness of the software development) without a change in the syntax of the programming language itself.

Usually class libraries are developed taking into account the specific character of the given object-oriented programming language. Many OO programming languages (for example, C++, Eiffel, and SmallTalk) have such libraries. The library approach is a low-level approach, since the developer remains responsible for many concurrency pissues such as resource management and synchronization), which require professional knowledge in this area, and are time intensive to develop.

The main merit of the library approach is its low-level flexibility. The approach is thus often used where low level system or embedded programming is required. However, the approach does not address the problem of the complexity of concurrent software development (Bruno 1993). What we need is the ability to program at a higher level of abstraction.

2.2 The Integrative Approach

The integrative approach introduces new concurrent constructs into the syntax of the OO language, which facilitate concurrent programming. These constructs then hide the details of how the parallel implementation is actually achieved (Wegner 1990).

There are several methods for integrating object-oriented programming and concurrent processing: active objects, synchronized objects and distributed objects.

2.2.1 Active objects

An active object integrates the concepts of an object and a process. An active object is a standard object, with attributes and methods, which also has its own thread of calculations, i.e., its own actions. Active objects can support two types of parallel calculations: introobject and inter-object. Depending on what type of parallel calculations is implemented, active objects can be of the following types (Wegner 1990):

- *Serial*. Active objects of this type can process only one message at a time. In other words, these objects do not use internal parallel processing. Languages using serial active objects are POOL (P.H.M. America: A. Yonezawa and M. Tokoro 1987) and Eiffel// (Caromel 1990);
- *Quasi-concurrent*. In such active objects several methods of activation can exist simultaneously, but only one of them is in the state of execution. This approach is used in the languages ABCHL/1 (Yonezawa 1986) and ConcurrentSmallTalk (Tokoro 1987);
- *Concurrent*. Active objects of this type allow parallel calculations inside the object itself, i.e., processing several queries simultaneously. In this case a certain degree of control of the execution, determined by the programmer, can be present. Among the languages, which use concurrent active objects are CEiffel (Lohr 1993) and ACT++ (Kafura 1990);

According to a key principle of object-oriented programming, an object must at the very least be reactive, i.e. react to events or messages. Active objects not only react, but also have their own thread, which is started immediately after the creation of the object. Thus, two types of active objects are distinguishable: *reactive* active objects and *autonomous* active objects. The first correspond to the principle of reactivity and are activated only on receipt of a message (ACT++, CEiffel), whereas the second type can independently execute in addition to responding to events (POOL, Eiffel//).

Another detail concerning the reactivity of active objects is the method for message acceptance. There are two methods for message acceptance: *explicit* and *implicit*. In the explicit method, the object is forced to accept all messages it receives (although its execution can be postponed). Implicit acceptance means that the object may refuse to accept a message according to some rules or constraints.

As an example of implicit acceptance, many languages (e.g. POOL and Eiffel//) have autonomous active objects with the notion of a 'body'. A 'body' indirectly describes the types and a sequence of queries that the object will accept during its activity. Eiffel// has a class PROCESS. An active class is a subclass of PROCESS. These objects have a routine ‘live’, which is the 'body' of the object. This function is defined in class PROCESS. However, to give it specific functionality, it is usually overridden in the subclasses. Other features of class PROCESS make it possible for the active object to manage the acceptance of calls in the ‘live’ feature.

2.2.2 Synchronized objects

Synchronized objects represent a further level of integration, in which synchronization is associated with the creation of objects. Messages are the explicit mechanism of synchronization between the sending object and the receiving object. The literature discusses two levels of synchronization: synchronization at the Message-Passing Level and synchronization at the Object Level. The difference is best illustrated via an example.

Assume there are two objects: *sender* – the object, which sends the message, and *receiver* – the object to which this message is addressed. There are two possible interaction behaviours for these objects. In the first of them, called synchronous transfer, *sender* blocks until the *receiver* completes execution of the message.

In the case of active objects, the sender and receiver execute independently of each other. This leads to the possibility of using asynchronous communication. The sender does not block; instead, it sends the message and then continues its execution. This type of object interaction can be implemented in different ways. One approach involves separating the call from the waiting object. Only when a calling object requires a result (to perform some actions on it) is synchronization with the called object required. This is known as Wait-by-necessity, implemented in the Eiffel// language (Caromel 1990).

Synchronization at the object level is of three types: intra-object synchronization, behavioral synchronization and inter-object synchronization.

In the case of intra-object parallel processing (in which the object simultaneously processes several requests), it is necessary to monitor the operations in order to guarantee the state of the object. Usually control is achieved by mutual exclusion between the operations. Intra-object synchronization can be illustrated with the "readers-writers" problem. All the existing readers can simultaneously access the shared book but the presence of one writer excludes access for all readers and writers. The shared book would be responsible for ensuring mutual exclusion, i.e. only one writer at any one time.

In behavioral synchronization, an object delays until a condition is met, instead of reporting an error. For example, in a bounded buffer, the buffer accepts values until it becomes full. When it is full, it simply waits until a value is removed, at which point it can insert the next value. Inter-object synchronization is used when it's necessary to synchronize the interacting objects.

To implement these methods of synchronization, different models of concurrency have been developed, which are subdivided into centralized (synchronization is achieved at the object level) and decentralized (synchronization is achieved at the method level) models.

An example of the use of the centralized synchronization model is Procol (Van den Bos 1991). The Path Expressions concept is implemented in this language, where the interleaving of invocations is determined with the aid of a special notation.

The *body* concept (discussed earlier) is another example of the centralized synchronizing model. However, the use of the *body* concept has difficulties associated with its implementation. This is due to the fact that in some situations the *body* can describe both the behavior specific to the application and the logic for accepting invocations. Taking into account that invocations are managed in a centralized way, and also that the *body* by its nature is defined imperatively, a number of problems have been raised concerning its implementation (Lohr 1993).

Another implementation of a centralized synchronization model is Behavioral Replacement. This model is used within the framework of the Actor language (Agha 1986). An actor has a mail address and a behavior. The mail address of an actor may be freely communicated – a feature which results both in the ability to reconfigure the system, and in the ability to extend a system (since mail addresses from the outside may be communicated). In response to processing a communication targeted to an actor, the behavior of an actor consists of three kinds of actions. An actor may send communications to specific actors it knows the mail address of. In particular, an actor may send communications to itself. An actor may create new actors. Initially, the mail address of such actors may be known only to the creator and possibly to the actor itself. However, the mail address can be subsequently communicated. An actor must specify a replacement, which will accept the next communication. The replacement may process the next communication even as other actions occurring as a result of processing the previous communication are still being executed. This model implies intra-object parallel calculations and synchronization.

The combination of Behavior Replacement and behavioral synchronization (when the active object appears serial) leads to the concept of abstract states. If one has a bounded buffer, we might need three abstract states: *empty*, *full* and *partial*. The abstract state of *partial* within the framework of this concept is expressed with the aid of the union of the states of *full* and *empty*. After the object processes the query, the next abstract state is calculated so that if it is possible to renew the state and the accessibility of the services of the object. The ACT++ language is an example of this concept (Matsuoka 1993).

The decentralized synchronization model is implemented via Guards, Locks or Annotations.

In the case of Guards, each feature of the object has a guard (or Boolean condition) associated with it for the object to become activated. The use of guards is convenient with the integration approach, since synchronization expressions need not be placed in the object. Actions are blocked or unblocked explicitly. However, this model of execution appears relatively slow. An example of the use of this synchronization model is the Guide language (Voss 1999).

An example of the use of Locks is to be found in the Java language. To synchronize threads, Java uses monitors, which are a high-level mechanism for allowing only one thread at a time to execute a region of code protected by the monitor. The behavior of monitors is explained in terms of locks. There is a lock associated with each object. The *synchronized* statement performs two special actions relevant only to multithreaded operation: (a) after computing a reference to an object but before executing its body, it locks a lock associated with the object, and (b) after execution of the body has completed, either normally or abruptly, it unlocks that same lock. As a convenience, a method may be declared *synchronized*; such a method behaves as if its body were contained in a synchronized statement[2].

The Java Virtual Machine allows an application to have multiple threads of execution running concurrently. There are two ways to create a new thread of execution. One is to declare a class to be a subclass of *Thread*. This subclass should override the *run* method of class *Thread*. An instance of the subclass can then be allocated and started. The other way to create a thread is to declare a class that implements the *Runnable* interface. That class then implements the *run* method. An instance of the class can then be allocated, and passed as an argument when creating a thread.

There is also a concept, where two locks are associated with an object: one for the *reader* methods, and another – for the *writer* methods. This concept is used in the Distributed Eiffel language (Gunaseelan 1992), which is a modification of the Eiffel language. Any operation can be declared as `ACCESSES` (for the reader methods) or `MODIFIES` (for writer methods). If any of these declarations are present, then the operation must obtain the read or write lock for the object before it will be able to begin its execution. Locking will not be achieved without those qualifiers.

Another modification of the Eiffel language is CEiffel (Lohr 1993), which uses the synchronization model called Annotations. In this language it is possible to determine

---

[2] http://java.sun.com/docs/books/jls/first_edition/html/17.doc.html

the binary symmetrical relation of compatibility between the operations of an object. If one operation is declared as compatible with another, then such operations can be executed in an overlapping manner (they can use the same resources). Incompatible operations are by definition mutually exclusive.

2.2.3 Distributed objects

The third level of integration of parallel calculations into the object-oriented languages of programming is a distributed object. This level of integration assumes that an object can be a distributed module, which can be distributed or replicated among several processors. To make the program able to carry out its parallel calculations concurrently, this program must be implemented with a multiprocessor or a multi-computer network. Some approaches using distributed objects are discussed below.

EPEE (Jezequel 1993) uses parallel calculations for the data of the type SPMD (single-program, multiple-data). The large structures of data, utilized in the EPEE language, are divided into fragments, which are distributed together with the replicated code between CPUs of a multi-computer. Each CPU processes a fragment of data while interacting with others CPUs if necessary. The syntax of EPEE is identical to Eiffel.

Another language, which uses distributed objects, Charm++, supports both parallel calculations for SPMD type data and parallel processing of the type MIMD (multiple program/data). In this language, reactive active objects are used. To define such an object, the keywords ‘chare class’ are used. There is also a version called 'branched chare class'. The code of this class is replicated between the nodes of a computer network and each of the nodes performs operations on a certain fragment of the replicated object.

The 'branched chare class' interface reflects fragmentation by describing the messages, which it can accept data from other fragments, and also from external fragments. Overall, Charm++ reaches a higher degree of integration by comparison with EPEE. Charm++ is an extension of C++.

The majority of the approaches mentioned in this subsection require syntactic changes to the associated programming languages. These approaches assume the use of a number of the keywords, connected with the implementation of parallel computation, in the declarations of objects and methods. As explained in chapter 1, SCOOP adds only one keyword to the Eiffel language. This issue will be explained in more detail at the beginning of Chapter 3.

2.3 The Reflective Approach

We explained earlier in this chapter that the Library Approach is more suitable for low-level system programming, while the integrative approach is useful in applications. The Reflective Approach attempts to combine the two, preserving the merits of each (the simplicity of the Integrative Approach and the flexibility of the Library Approach).

Reflection is a general methodology for describing, controlling, and adapting the behavior of a computational system. The basic idea is to provide a representation of the important characteristics/parameters of the system in terms of the system itself. The characteristics of static presentation and dynamic execution of applications are determined in one or several programs (which can be an interpreter, a compiler or other programs), which present the behavior of the system while doing calculations. Such programs are called meta-programs. Reflection fits especially well with object concepts, which enforce good encapsulation of levels and modularity of effects.

Based on the fact that the meta-programs are objects, this system is called meta-object protocol (Kiczales 1991).

Below are some examples of the Meta-Object Protocols (MOP) implementation. The CodA platform (McAffer 1995)] is the general reflex architecture, built on the objects and based on meta-objects. By default CodA is examining seven meta-objects, connected to each of the objects. These meta-objects are message sending, receiving, buffering, selection, method lookup, execution, state accessing. The object, which has default meta-objects, behaves as usual passive, sequential object. The connection of special meta-objects makes it possible to selectively change the specific aspect of the presentation model of idea or execution for a certain object.

Other two reflexive architectures, namely Actalk and GARF, are more specialized and propose smaller collections of meta-objects. The Actalk platform (Briot 1996) helps to experiment with different models of synchronization and communication for a predetermined program by changing different components: activity (for example, implicit or explicit acceptance of requests, intra-object concurrency), synchronization (for example, abstract behavior, guards), communication (for example, synchronous or asynchronous), invocation (for example, time stamp, priority).

The GARF platform (Garbinato 1994) for distributed and resistant to errors programming allows a wide variety of mechanisms around two components: object control and communication.

2.4 Another SCOOP-like Implementation

(Jalloul 2000) proposes a method for the integration of parallel processing into object-oriented languages called CSS (Communicating Sequential Systems). On the basis of this method, he created CEE (Concurrent Extension to Eiffel).

Similarly to Meyer's SCOOP, keyword *separate* is also used in CEE. However, in contrast to SCOOP, CEE provides critical regions and conditional critical regions, but does not rely on procedure calls and require conditions.

In CEE, a program is subdivided into many "internally concurrent sequential systems". These systems work in parallel. Each of them in this case can have internal parallel calculations. To wait for returned values, a wait-by-necessity mechanism is used. CEE has a kernel, implemented in the Eiffel language, which is located on the upper level of communication software for distributed processes. Thus the implementation of parallel processing is hidden from the programmer.

Based on the *separate* declarations, the compiler divides an Eiffel program into several systems, each of which then is compiled by the Eiffel compiler. During the execution, interaction with other systems is translated into the queries to the kernel, which are then sent to the controller of the matching system.

2.5 The inheritance anomaly

The term *inheritance anomaly* was coined in 1993 by Matsuoka and Yonezawa (Matsuoka 1993) to refer to the problems arising from the interweaving of behavioural and synchronization code in descendant classes.

For example, consider class BUFFER (Appendix G) that has a function routine *item* that returns the oldest element in the buffer. In modern languages such as Java and C#, the burden of enforcing the synchronization constraints must ultimately lie with the buffer itself. Suppose we have a new class BUFFER2 (Appendix G) that inherits from BUFFER. In this new class we would like to define a new function *item2* that works like *item*, except that it cannot be executed immediately after a call to *item*. In Java and C#, not only must the behaviour of *item* be redefined (e.g. by introducing a history variable), but this redefinition must be intertwined with synchronization code. This interweaving of behavioural and synchronization code makes such programs difficult to develop and understand.

According to (Milicia 2003), SCOOP also suffers from the inheritance anomaly. However, Meyer in (Meyer 1999) disagrees. Meyer appears to be correct in this regard. It is true that in SCOOP, *item* must be redefined, but only behaviourally. No synchronization code is needed at all (as shown in detail in Appendix D). In fact, BUFFER can be a regular Eiffel class. If it is needed as a concurrent buffer, it can be declared as a `separate` supplier, and the preconditions of *item* and *item2* immediately become wait conditions. However, in Java and C#, *item* must be redefined both behaviourally and with synchronization code (using *synchronize* and *throw*/*catch*).

Chapter 3

Simple Concurrent Object-Oriented Programming

In this chapter, we review the framework developed by Meyer (Meyer 1997), called Simple Concurrent Object-Oriented Programming (SCOOP). Meyer's notation will be used to describe SCOOP. Compton (Compton 2000) developed a prototype implementation of SCOOP, including a run-time system. Compton also contributed new notations and refinements of existing concepts, which assist in the implementation of SCOOP in practice.

SCOOP is an extension of Eiffel that allows for parallel object-oriented calculations by adding a single reserved word `separate` into the syntax. Meyer makes an interesting claim: a single new keyword (`separate`) provides for a full-fledged concurrency mechanism. A general rule of software construction is that a semantic difference should always be reflected by a difference in the software text (Meyer 1997).

A SCOOP compiler (or in our case the Generator) will translate the `separate` constructs into target code according to the SCOOP model. Even though the SCOOP model uses only one extra keyword "separate" to take care of all the concurrency issues, separate has a different semantic meaning when used with class declarations, attributes, and routine parameters. In the sequel, the model will be described along with various aspects of the SCOOP mechanism for Eiffel. We also describe problems arising with the model, and possible solutions.

In chapter 4 we will describe the implementation of the SCOOP-to-Eiffel code Generator and we will discuss the implementation of various SCOOP elements into Eiffel target code.

3.1 Processors and Subsystems

One of the key concepts of SCOOP is the *processor*. As shown in Figure 3-1, a computation is performed by a *processor* that applies certain *actions* (or routines) to certain *objects*. In the sequential case, there is only one processor. In the concurrent context, we have two or more processors. This is what concurrency is all about and can be taken as the definition of concurrent processing.

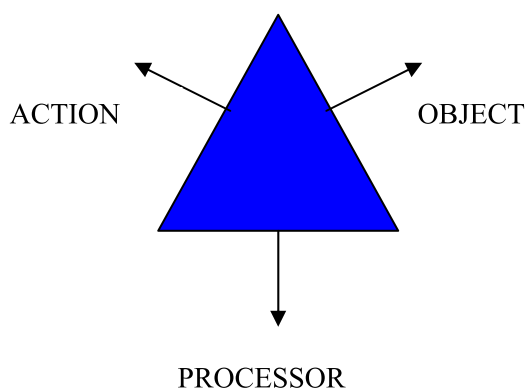

Figure 3-1: Processors

According to Meyer's definition, a *processor* is an autonomous thread of control capable of supporting the sequential execution of instructions for one or more objects (Meyer 1997, page 964).

This definition assumes that the processor is some device, which can be implemented as hardware (e.g. a computer equipped with its own central processor), or as software (e.g. a thread, task or stream). The given definition describes an abstract processor and enables the system to use as many actual processors as required.

A *subsystem* is the 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 (Compton 2000, page 18).

A *separate object* is any object that from the viewpoint of one object is in a different subsystem. At run time, any `separate` object can only be referenced (if reachable at all) through a `separate` entity (Compton 2000, page 19). An entity is either an attribute of a class, a formal argument of routines, 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 (Compton 2000, page 20).

A *separate call* is any routine call `x.f (...),` from the current object in which the call is made, where the target, `x`, is a `separate` object (Compton 2000, page 20).

A subsystem is created simultaneously with the creation of a `separate` objects and executes the object's instructions. Several processors that run different `separate` objects allow concurrent execution. Processors may themselves contain subprocessors.

`Separate` objects, in turn, can create objects. Those objects can be shared with other processes; they can receive references to the objects that are carried out by other processors. Thus the processor can carry out operations not only on one `separate` object, but also on a set of objects. While sequential Eiffel contains one subsystem, the use of SCOOP provides an unlimited number of subsystems.

A new subsystem is created with the creation of a `separate` object. Non-separate objects are created in the same subsystem as the object that has created it. Thus, it will be considered as a `separate` object by other subsystems. Any object (whether `separate` or non-separate) can belong to only one subsystem. For communications (connections) between the objects that take place in different subsystems, `separate` references are used. Fig. 3.2 illustrates a SCOOP runtime system consisting of a number of subsystems.

3.2 Routine calls in sequential Eiffel

Feature call in sequential Eiffel is defined as follows:

```
x.f(a)
```

i.e., execute routine `f` with argument `a` on target `x`

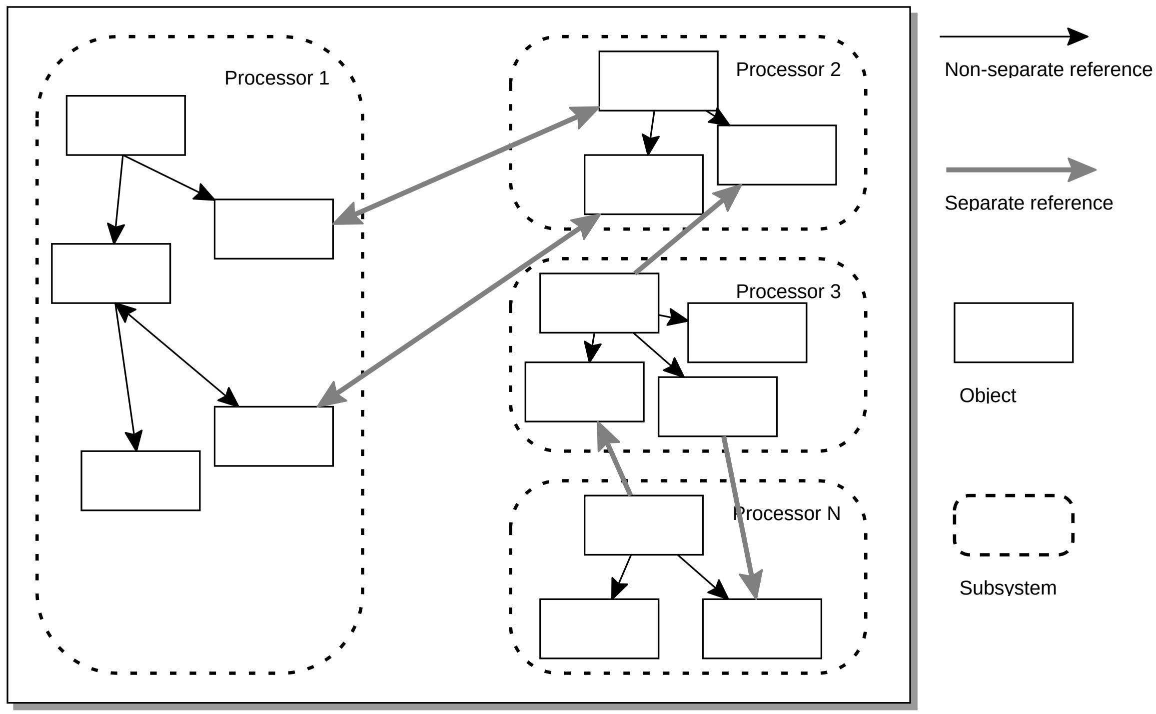

Figure 3-2: SCOOP System

According to the semantics of object-oriented programming, we can distinguish two types of procedure calls:

- Case 1: A *command* feature call

```
x.c (a)
```

- Case 2: Assignment involving a *query* call

```
y := x.q(a)
```

In Eiffel, there is a strong distinction between a *command* and a *query*. A query has a corresponding type; a command does not. Consequently, a command is an independent statement, while a query is on the right hand side of an assignment.

In Case 1, `c` is a command. The command is executed on the target `x`, and when completed, processing continues at the instruction following `x.c(a)`.

In Case 2, since the assignment statement has `q` on the right hand side, `q` must be a query. Execution switches to the object attached to `x`, and when completed, the result is assigned to `y`. Execution then continues at the instruction following the query.

There is only one processor, and therefore only one subsystem. This processor executes the current routine as well as the routines `c` and `q`.

The syntax of Eiffel does allow queries to change the state of the object on which it is called. However, this is discouraged in practice since contract checking would cause the state of the checked object to change. Therefore, in Eiffel there is a strong semantic separation between a command and a query. While a command changes the state of an object, the query should not.

3.3 Routine calls in SCOOP

Generalizing program execution to concurrent object-oriented models requires a change in the feature call definition.

Suppose as before that `x` is a `separate` entity. There must then exist at least two subsystems, the current subsystem in which the code `x.c(a)` occurs, and the subsystem associated with `x`. The call to `x.c(a)` will be executed in the latter subsystem, and the current object making this call continues executing in its own subsystem.

In the case of the assignment `y := x.q(a)`, the current system blocks while the subsystem associated with `x` executes query `q` to completion (called *wait-by-necessity* as explained in chapter 1).

Commands may be executed in parallel as different processors process them. A query may need to return a result before the program can continue. For example, in the code below the query `q` is called on entity `x`:

```
y := x.q (a)
…
z := y + 1
```

Execution does not continue until the result is computed and assigned to `y`.

Caromel (Caromel 1989) was the first to define the notion of "Wait by Necessity". In the original definition, some cases were allowed which enabled the continuation of parallel calculations even in the case of a query. For example, in the code fragment above, entity `y` is not used until later in the statement `z := y +1`. Thus, we could wait until `y` is actually used before synchronizing with the other subsystem.

(Compton 2000) implemented Caromel's proposal, but the resulting implementation turned out to be inefficient. In this thesis, we follow the original proposal of SCOOP (Meyer 1997), and thus, the calling subsystem always waits at `y:=x.q(a)` before continuing to execute. This is much simpler to implement than Caromel's proposal.

Having considered existing types of calls and the ways they can be processed in the SCOOP program, it becomes evident that there can be two options for feature calls. In the first case in which the target is not separate, the execution of a call is made by the same processor that executes the calling object (the object on behalf of which the call is made). In the second case in which the target is separate, some other processor processes the call (not the one that processes the calling object). To specify how and where it is necessary to execute a call, some syntactic construct is required to reflect the semantic intentions in the text of the program. The syntactic keyword `separate` is used to reflect this semantic difference.

3.4 Eiffel Syntax and Semantics for SCOOP

According to Meyer, for the SCOOP implementation in Eiffel, it is necessary to add only one keyword `separate`. The object declaration as `separate` specifies that a new processor will execute it. Possible ways of applying the `separate` keyword and syntax patterns are presented in figure 3-3.

A class can be declared as `separate` as follows:

```
separate class TY
```

Figure 3-4 will help to understand the semantics of SCOOP using the syntax patterns presented in figure 3-3.The declaration

```
x : TX
…
create x.make
```

means that object *O1* of type `TX` will be created in Root Subsystem (*Ho*) and entity `x` is attached to it. The declaration

```
y: separate TY
…
create y.make
```

means that the entity `y` is attached to objects whose routines are executed by other processors. Another subsystem *Hy* is created. Object *O2* of type `TY` is created in subsystem *Hy*. Entity `y` is attached to object *O2*.

```
separate class TY
  x : TX
  y : separate TY
```

```
z : separate TZ
c ( a : separate … )
    is
        do
    …
        end
    …
    r is
       do
         create x.make
         create y.make
         create z.make
         x.c1 ( a1 )
         y.c2 ( a2 )
          x := x.q3 ( a3 )
          y := z.q4 ( a4 )
          c(a5)
        end
```

Figure 3-3 : SCOOP Syntax

According to (Meyer 1997, page 967) all three qualifiers used at the declaration of classes (*separate*, *expanded (*the values are objects*)* and *deferred* (classes that leave the implementation of some of their features entirely to proper descendants)) are mutually exclusive. This follows directly from the sense of the appropriate qualifier and from the semantics of the Eiffel language. Descendant classes do not inherit these qualifiers. Thus such a declaration is invalid if `TY` is already declared *expanded* or *deferred*.

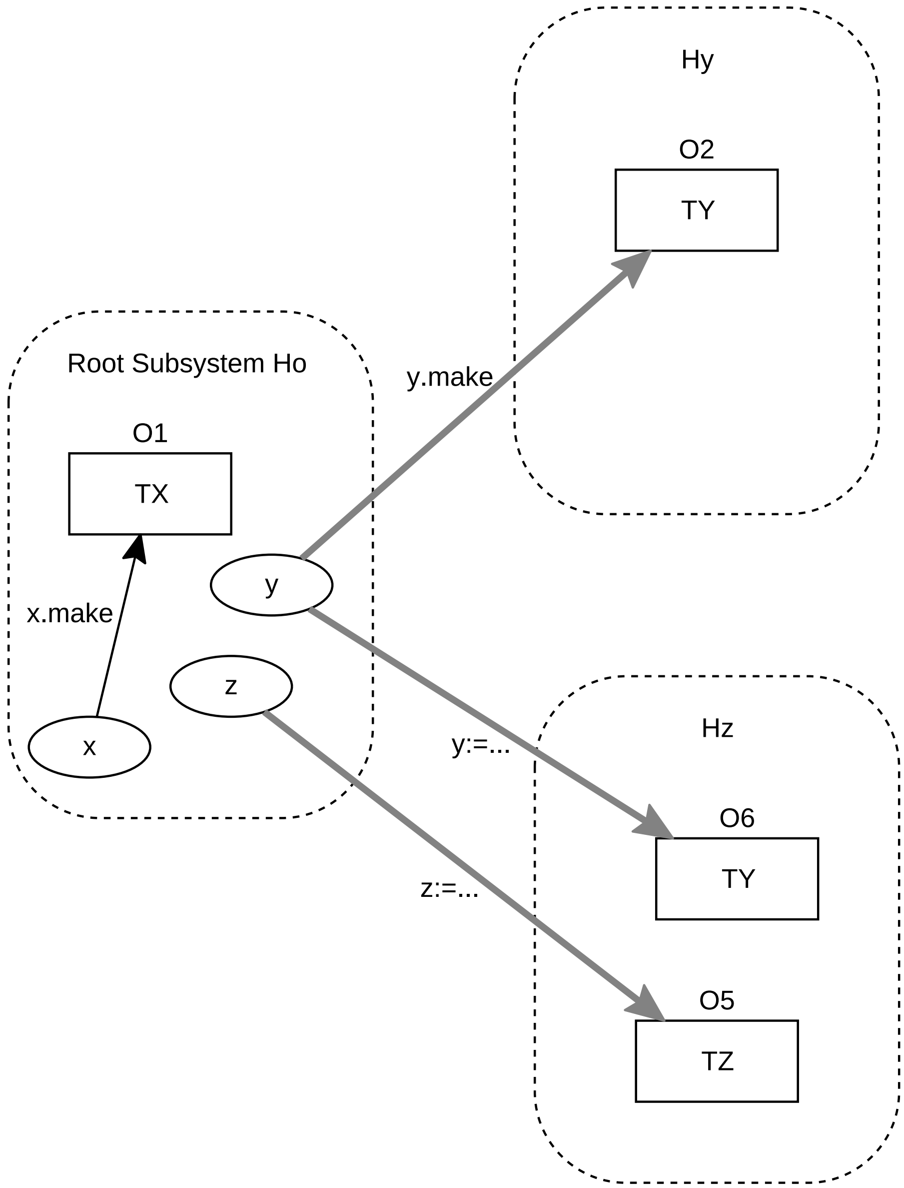

Figure 3-4: SCOOP Semantics

The statement

```
x.c1(a1)
```

means that subsystem *Ho* performs command `c1` with an argument `a1` on object *O1*.

The statement

```
y.c2(a2)
```

means that subsystem *Hy* performs command `c2` with an argument `a2` on object *O2*.

The statement

```
y := z.q4(a4)
```

means that subsystem *Hz* executes query `q4` with an argument `a4` on object *O5*. Object *O6* is created (this is the result of the query) and entity `y` is attached to it.

The statement

```
c(a5)
```

means that subsystem *Ho* gets unique access and locks the `separate` object attached to `a5` and executes the command `c` through to completion after which it releases the lock.

3.5 Separateness consistency rules

A problem arises when a non-separate object is used in place of a `separate` object. This non-separate object is known as a *traitor* object. Meyer introduced four separateness consistency rules. These rules guarantee that no traitor object situation can occur. The Generator must flag any traitor as a compile time error. The rules are listed below.

The **separateness consistency rule (1)**: If the source (`y`) of an attachment in an assignment instruction (or equivalently, argument passing) is `separate`, its target entity (`x`) must be `separate` too. In practice, this means that if we have the entities declared as

```
x: SOME_TYPE
y: separate SOME_TYPE
```

then operations such as `x:=y` are forbidden.

For example, suppose we allow `x:=y` in the above case. The compiler assumes that `x` is in `y`'s subsystem as `x` is attached to the same `separate` object that `y` is attached to. Thus when `x.c` is executed, the precondition of `c` may (incorrectly) be treated as a wait condition rather than a correctness condition. Thus the object to which `y` is attached is now a traitor.

Separateness consistency rule (2): If an actual argument of a `separate` call is of a reference type, the corresponding formal argument must be declared as `separate` (Meyer 1997).

Assume we have the declarations in figure 3-5. Different processors handle objects `x` and `y`. Having declared `x` and `arg` as non-separate we have created a situation in which the subsystem of `x` will treat `y` as a local (i.e. non-separate) object. But this is wrong because `y` is really in a different subsystem. Hence, it is necessary to declare `arg` as `separate`.

Separateness consistency rule (3): If the source of an attachment is the result of a `separate` call to a function returning a reference type, the target must be declared as `separate` (Meyer 1997).

```
class FIRST_CLASS feature
      y: SOME_TYPE            -- non-separate
      x: SECOND_CLASS   -- x is declared 'separate'
      some_feature is
       do
            x.f(y)
       end
end
```

```
separate class SECOND_CLASS feature
      f (arg: SOME_TYPE) is
       do
                   . . .
       end
 end
```

Figure 3-5: SCOOP Separate consistency rules

The third rule means that in a `separate` call, the reference to the returned value can be placed only in a variable described as separate.

Separateness consistency rule (4): If an attachment or the result of a `separate` call is of an expanded type, its base class may not include, directly or indirectly, any non-separate attribute of a reference type (Meyer 1997).

This rule means that we can pass an expanded object as an argument in a `separate` call, only if such expanded objects have no references to other objects. Non-observance of this rule can result in the occurrence of a traitor. This will result in the compiler treating this call mistakenly as a synchronous local call, while the attached object is `separate` and needs to be handled asynchronously.

3.6 Synchronization in SCOOP

In his work, Meyer considers various existing mechanisms of synchronization and their applicability in the context of parallel object-oriented calculations. For use in the SCOOP mechanism, Meyer describes a method for synchronized access to shared objects, which does not contradict the principle of inheritance, works well with DbC, and also guarantees that actions on objects are made in the sequence that we expect.

In SCOOP, at each moment of time there is at most one executing routine on a given object. Furthermore, synchronization is carried out at the level of an object instead of at the level of its entities (attributes or variables). Also, a subsystem executes calls to it from other subsystems in the order received.

Consider the following example (Meyer 1997). Suppose we have a requirement to remove two consecutive elements from a shared structure `buffer` (see chapter 1). To remove one element, procedure `remove` is used. For a double remove, we might choose to write:

```
…
buffer.remove
buffer.remove
…
```

However, between these two calls, another object can obtain access to the buffer and execute any actions on it. Hence, it is impossible to guarantee that those two required elements will be removed.

To solve this problem in SCOOP, it is necessary to write down the two consecutive calls inside one procedure (to encapsulate them) and to pass to the procedure the reference to the shared object.

```
remove_two (buffer: separate BUFFER) is
      do
            buffer.remove
            buffer.remove
      end
```

Figure 3-6: removing elements from buffer using the SCOOP execution model

In this case the buffer will be inaccessible to other clients until the termination of of the body of `remove_two`. This behaviour results in the following SCOOP rule:

Separate Call rule: The target of a `separate` call must be a formal argument of the routine in which the call appears (Meyer 1997, page 985).

As another example, suppose we want to call feature `put` on a `separate buffer` we then write the code for `buffer_put` as shown below (instead of `buffer.put(…)`):

```
buffer_put (some_buffer: separate BUFFER) is
    ...
    do
     -- calling put on some_b
     some_b.put(…)
    end
```

Figure 3-7: adding elements to buffer using the SCOOP execution model

3.7 Semantics of preconditions as wait conditions

As described earlier, in the case of sequential execution, preconditions work as expected as a correctness condition. In SCOOP, however, the precondition is no longer a correctness condition but a wait condition.

Consider a situation in which we have three subsystems S1, S2 and S3. Suppose S1 calls a routine *r* in S2. S2 checks the precondition of *r* and then executes the body of *r*.

The problem is that in between the evaluation of the precondition and the execution the body, another subsystem may falsify the precondition. This situation has been named the *concurrent precondition paradox*. Suppose routine *r* is as follows:

```
      -- subsystem S2

      a: separate TYPE

      r(x1: separate TYPE1, x2: separate: TYPE2; x3: TYPE3) is
            require
              x1_validity: x1 /=Void
              x2_validity: x2 /= Void
              a_validity:  a /= Void
              x3_validity: x3 /= Void
            do
              -- routine's body
            end
```

The precondition clauses `x1_validity`, `x2_validity` and `a_validity` are called *separate preconditions* as they have occurrences of the routine arguments or class attributes that are declared `separate`. The non-separate precondition `x3_validity` remains a correctness condition (if false an exception is immediately raised).

By contrast, the subsystem must gain a lock on all the `separate` entities before checking the `separate` preconditions. If these preconditions evaluate to true, the body is executed and then the `separate` entities are unlocked. If the separate preconditions evaluate to false, then the separate entities are unlocked, and the separate preconditions rechecked at some subsequent time. We thus have the following constraint:

Separate call semantics: Before a `separate` call can start executing the routine's body, the `separate` call must wait until every blocked object is free, and every `separate` precondition clause is satisfied.

Chapter 4

SCOOP to Eiffel+Threads Code Generator

The purpose of this chapter is to develop a Generator that will translate a SCOOP program (that uses the `separate` keyword) to code in Eiffel+Threads (as described in section 1.1). In order to develop the Generator, the development of an appropriate mapping from SCOOP programs to generated Eiffel+Threads code is required. While (Meyer 1997) provides a comprehensive overview of the proposed SCOOP functionality and use, no implementation details are provided. In this chapter we develop and describe the mapping that the Generator uses to do the translation. The mapping must be done in such a way as to obey the SCOOP model developed in the previous chapter.

SCOOP functionality includes

1. Declaration and instantiation of `separate` objects;
2. Call of features on `separate` objects;
3. Argument passing (expanded and reference types);
4. Exclusive locking of single and multiple `separate` objects;
5. Declaration of `separate` features including both attributes and routines;
6. Wait conditions and DbC;
7. Wait by necessity;
8. Support for distributed execution and Concurrency Control Files (CCFs).

The Generator fully implements items 1 to 7. The cross-platform multi-threaded Eiffel+Threads runtime does not support distributed execution, which means that 8 is not implemented by our Generator. Thus, as far as we are able to ascertain, the Generator is currently the most complete implementation of SCOOP.

The other SCOOP implementations (Compton 2000; Meyer 2003) also do not support distributed execution, although the intention is to ultimately support distributed execution in (Meyer 2003). (Compton 2000) does not support wait conditions and DbC (item 6), and (Meyer 2003) does not yet support locking of multiple `separate` objects (item 4).

As described in the first chapter, Eiffel+Threads is standard Eiffel together with a cross-platform threads library for concurrent execution. A BON diagram for the Thread library is shown in Figure 4-1, and Appendices A, B and E contain more details.

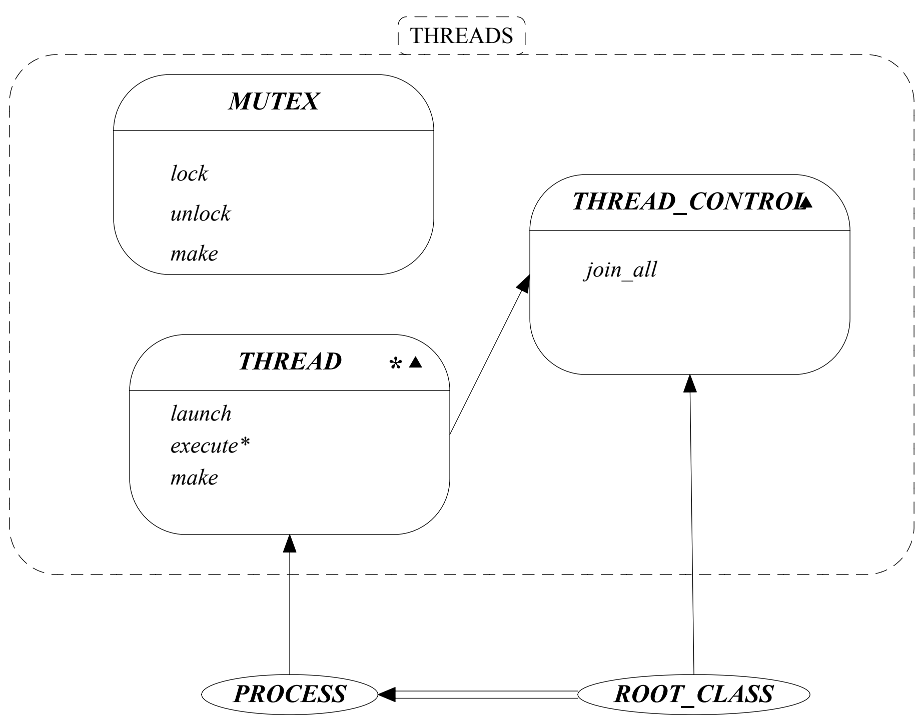

Figure 4-1 BON diagram of Threads library

Suppose we have a ROOT_CLASS that launches two threads *p1* and *p2* each of type PROCESS. The ROOT_CLASS does this in its creation procedure as follows:

```
p1.launch
p2.launch
join_all
```

The PROCESS threads must effect *execute* inherited from class THREAD (figure 4-2). A *launch* invokes the implemented *execute* routine. When all the *execute* routines terminate, then *join_all* terminates the system. Class MUTEX supports data locking in the standard way. A mutex can be created to protect data. Routine *lock* waits until access is granted and *unlock* frees the mutex to other threads. It is assumed that the underlying OS implementation of mutexes is fair (an assumption that has been verified under Windows and Linux).

4.1 Eiffel SCOOP project files

As explained in the previous chapter, the keyword separate is used as follows:

```
separate class A_CLASS …

a_entity : separate A_TYPE …

another_routine (separate_argument: separate SOME_TYPE) is
    require
       separate_argument …
```

The Generator is invoked on an Eiffel Scoop Project file (with extension ".esp"). Consider the producer-consumer example described in Section 1.2. Classes ROOT_CLASS, PRODUCER and CONSUMER all have occurrences of the separate keyword in their text. Class `BUFFER` does not have any occurrences of the `separate` keyword.

The text of each class `C` that has occurrences of the `separate` keyword is placed in a file `c.es`. The Generator will automatically transform each SCOOP class `C` to a new generated class `CG`, and the text of `CG` is placed in a file `c.e`. The Eiffel SCOOP Project file for the producer-consumer example is as follows:

```
root   root_class.es
       consumer.es
       producer.es
```

The project file does not specify file `buffer.e` as the keyword `separate` does not appear in it. The root class is distinguished from the other classes in the project file. This is because:

1. the generated `root` class must inherit from `THREAD_CONTROL` which does not have routine `execute`. Instead of `execute`, the creation procedure of the `root` class is initially called. At the end of the creation procedure, `join_all` must be invoked for the system to exit;

2. the generated root class will have the responsibility for managing a global shared integer variable called `requests_pended`, that keeps track of the total number of `separate` calls across all subsystems. This variable is used by all the subsystems to determine when to safely exit (as will be explained in the sequel).

All other classes in the project file inherit from `THREAD` and effect `execute`.

4.2 Implementing Subsystems

Suppose we have SCOOP classes as follows:

```
separate class A_CLASS feature
     c1: C1
     c2: separate C2
     …
     create c1.make
     create c2.make
     …
end

separate class C1 … end

class C2 … end
```

Conceptually, `c1` and `c2` each have their own subsystem. However, it is usually the case that non-separate classes such as C2 are usually passive data containers, such as BUFFER in the producer-consumer example. The main requirement is that any call to such classes must run atomically in order to be protected from interference by other threads. They do not really need their own thread. By contrast, classes such as C1 are independent processes that must run in their own thread (e.g. the producer, or consumer).

The Generator therefore treats these two cases differently. For each entity such as `c2`, we merely declare an associated mutex `c2_mutex`. Any feature call `c2.f` is always "wrapped" with a lock to the mutex:

```
c2_mutex.lock
c2.f
c2_mutex.unlock
```

Thus, in the generated code, any routine with `c2` as an argument must also be passed `c2_mutex` at the same time. Such a procedure guarantees that the shared data object associated with `c1` is only accessed by one routine (atomically) at a time.

By contrast, `c1` must execute in its own thread. To implement `c1`'s subsystem, we must therefore proceed differently. Let *C1G* denote the generated code associated with the SCOOP class *C1*.

1) Each generated class such as C1G (other than the `root` class) inherits from `THREAD` and effects `execute`.

2) Each instance of C1G has its own buffer `request_buffer` which is a queue of `separate` calls (to routines of `C1G`) coming from other subsystems. Other subsystems must first obtain the lock to the buffer (called `request_buffer_mutex`) before being allowed to queue its routine call request. Each addition to the buffer increments by one a global integer variable `requests_pended` (which has a corresponding lock `request_pended_mutex`). Thus, at any moment in time, the value of `requests_pended` is the number of `separate` buffered calls across all subsystems.

3) The deferred feature `execute` (of `THREAD`) is implemented in `C1G` by repeatedly requesting a lock on the `buffer` and executing the oldest routine call request (say for routine `r`). When `r` (and all its sub-calls)

terminates, then `requests_pended` is decremented by one to indicate that there is one less call request to process. The global variable `requests_pended` must be zero before `execute` terminates, thus terminating the subsystem.

4) If any of the sub-calls of `r` is itself a `separate` call to some subsystem, then the same procedure is followed, i.e. the sub-call is registered with the buffer of requests for that subsystem, and that subsystem must complete the sub-call before control is returned to `r`. Thus `requests_pended` will not reduce to zero until every `separate` call (and its `separate` sub-calls) has been handled by the appropriate subsystem.

Steps 1-4 ensure (a) that every `separate` call is registered and executed, and (b) that subsystems only terminate when no more `separate` calls are possible.

The `root` class in the project file (say CR) has an associated generated `root` class called `CRG`. As described earlier, `CRG` inherits from `THREAD_CONTROL` and has the responsibility for managing the `requests_pended` global variable and its lock. The generated creation feature of class `CRG`:

1. initializes the `requests_pended` and its lock;
2. launches the appropriate threads. Any `create` statements in `CR` (e.g. **`create`** `c1.make`) involving `separate` calls (of type `C1`) must be followed by a `launch` in the corresponding generated code, i.e. (`c1.make; c1.launch`...). The launch command invokes the effected `execute` routine in `C1G`;

3. registers any `separate` calls in the creation feature of `CR` with the appropriate subsystems;

4. calls `join_all` for system termination.

There are thus three cases, each handled differently by the Generator:

a) the `root` class `CR`;

b) `C2` (i.e. passive data that must be protected);

c) `C1` (i.e. active processes that need their own threads in the generated code).

```
class SOME_TYPE

inherit
            THREAD
feature
            execute is
                  do
                        …
                  end

class SECOND_CLASS
     some_var: SOME_TYPE
     make is
                  do
                        …
                        create some_var.make
                        some_var.launch
                        …
               end
```

Figure 4-2 THREAD inheritance

Further experience with SCOOP may cause us to treat `C2` similarly to `C1`, but with a loss of the efficiency of the current model. The current generated code will run correctly with less thread overhead; it's only downside is that it over-serializes calls to passive `C2` type structures. Further experience with SCOOP is needed to evaluate which translation is better.

The Generator scans through all the classes mentioned in project file. The root class and other `separate` classes are each translated to generated code, as described in the overview presented above, and with further detail supplied in the rest of this chapter.

4.3 Effecting routine *execute*

Consider an instance of a `C1` type class. It is launched `(`by calling `launch)`, which in turn calls routine `execute.` It is the responsibility of `execute` to manage this thread (i.e. subsystem). Figure 4-3 shows a pseudo-code version of the effected `execute` routine. The body of the routine repeatedly accesses the oldest `separate` call to this subsystem (from other subsystems) in the request buffer, executes the call and then removes the call from the buffer. Thus `separate` calls are atomically processed in the subsystem in the order they are received (while other subsystems concurrently process their calls in the same manner). This is because any `separate` call must be registered with the subsystem, and only such calls are invoked by `execute`.

An example of the precise `execute` code is provided in Appendix C. The `stop_condition` involves an access to the global variable `requests_pended`, which is described in more detail below.

```
class C1G feature

      execute is
           do
                  from
                  until not (stop_condition)
                  loop
                    get_next_call_from_request_buffer
                    execute_call
                    …
                 end
            end
      …
end
```

Figure 4-3 the `execute` feature

4.4 Keeping track of separate calls

The `execute` routine in the generated code C1G needs to access the `separate` calls in the order received by this subsystem. Figure 4-4 illustrates the way in which this is done via a buffer `request_buffer` and routines to add a `separate` call to the buffer and remove a call (`set_feature_to_do, get_feature_to_do`). The request buffer is a list of TUPLE:

```
      request_buffer: LINKED_LIST[TUPLE]
```

Tuples are a mathematical cross product, implemented as an indexed linear data structure. The number of elements in TUPLE beforehand is not determined. Tuples are extremely useful in SCOOP, as no decorator classes are necessary to wrap the features, their arguments and other associated data. TUPLE will be used to store `separate` calls (e.g. the name of the call, and its arguments).

```
class C1G feature

      execute …

      request_buffer: LINKED_LIST[TUPLE]

      set_feature_to_do(feature_params_arg:TUPLE) is
          do
            requests_pended_mutex.lock
            requests_pended.copy(requests_pended + 1)
            requests_pended_mutex.unlock
            request_buffer_mutex.lock
            request_buffer.extend(feature_params_arg)
            request_buffer_mutex.unlock
          end

      get_feature_to_do:TUPLE is
          do
           request_buffer_mutex.lock
             if not request_buffer.is_empty then
              Result := request_buffer.first
             else
              Result := [Current,"NOTHING"]
             end
           request_buffer_mutex.unlock
          end

          …
end
```

Figure 4-4 A buffer to queue `separate` calls to a subsystem

`Separate` calls can be placed on the subsystem buffer using the routine:

```
set_feature_to_do(feature_params_arg: TUPLE)
```

Routine `set_feature_to_do` places the call data at the end of the buffer and is public (it can be called from any subsystem). If another subsystem wants to execute a `separate` call to this subsystem, it registers the call using `set_feature_to_do`. The other subsystem can then continue executing (for a feature that is a command), without blocking, and the `execute` routine of the current subsystem will get the call from the buffer and execute in the proper order. Queries and wait by necessity will be discussed in the next subsection.

Routine `get_feature_to_do` is called by routine `execute`:

```
get_feature_to_do: TUPLE.
```

It gets the oldest call (stored as a `TUPLE` on the buffer). Thus the buffer is organized under a FIFO scheme.

4.5 Command and function calls

`Separate` command and function calls to a subsystem are both registered in the subsystem buffer, as explained above. However, function calls (queries) are subject to the wait by necessity rule. In this section we discuss the differences between command and function calls.

4.5.1 Command Routines

Consider the SCOOP code for two subsystems `C1A` and `C1B` in figure 4-5. `C1A` makes a `separate` call request `f.some_command(arg)` to `C1B`. In `C1B`, the argument `arg` is declared as `separate`.

```
separate class C1A …
      c1b: C1B
      …
      c1b.some_command(arg)
      …
end

separate class C1B …
      some_command(separate arg: TYPE1) is
      …
end
```

Figure 4-5 Commands

As it currently stands, subsystem c1a directly calls and executes the separate routine some_command in subsystem c1b, and inappropriate interference with other threads might occur. Instead, we require that c1a register the call some_command(arg) with c1b's buffer of calls so that the c1b can later execute some_command atomically and safely in the appropriate order. The Generator must therefore map the call c1b.some_command(arg) in C1A to

```
          c1b.set_feature_to_do([Current,"SOME_COMMAND_STRING",arg1,
                                   arg1_mutex])
```

in the generated code C1AG. Recall that set_feature_to_do is a public routine in the generated code C1BG associated with subsystem C1B which is declared as

```
      set_feature_to_do(feature_params_arg: TUPLE)
```

The formal argument feature_params_arg is a TUPLE that can store the call c1b.some_command(arg) for later reference. The first field of the TUPLE stores the calling subsystem (i.e. Current that refers to c1a), the second field stores the name of the routine some_command as a string "SOME_COMMAND_STRING", and the third field stores the routine argument arg. Since arg is declared as a separate argument, it refers to a different subsystem, and thus any accesses to `arg` must be via it's lock `arg1_mutex`, which is passed in the fourth field of the `TUPLE`.

The first field of the `TUPLE` (`Current`) allows the `c1b` subsystem to access the global variable `requests_pended`. This variable must be incremented when the call is registered and decremented when it is executed.

After the information on a call is placed in the buffer, the subsystem associated with `c1a` continues execution while the subsystem associated with `c1b` will process the call.

4.5.2 Function Routines

In the case of a command routine, the calling subsystem registers the command call with the target subsystem, and then continues execution without blocking. In the case of a function call (see Figure 4-6), we have some version of the "wait by necessity" mechanism, as defined in Section 3.3. Thus, if the calling subsystem executes an assignment `x:=y.f`, then it must block until the subsystem associated with `x` has executed the call `f`.

As for commands, function calls are also registered with the target subsystem using routine `set_feature_to_do`. The function call is registered in the same way as a command with only one difference:

- An extra field of the TUPLE is reserved for the return value of the function.

```
 separate class CA feature
           f: SOME_TYPE is
                 do
                       …
                 end
           …
end

 separate class CB feature
           x: separate SOME_TYPE
           y: CA
           some_feature is
                 do
                   x := y.f
                   z:=x
                 end
      …
end
```

Figure 4-6 Function calls

The following code will be generated by Generator:

```
      class  CB
      …
      x: SOME_TYPE
      x_mutex: MUTEX
      y: CA
      y_mutex: MUTEX …
      some_feature is do  …
                   x_mutex.lock
                   y.set_feature_to_do([Current,"f_STRING", x, x_mutex])
                   x_mutex.lock
                   -- We acquire mutex second time and wait until
                   -- it will be released in CA
                   x_mutex.unlock
                   z:=x
                 end
      …
```

```
class  CA
…

execute is
      local
            f_return: SOME_TYPE
            f_return_mutex: MUTEX
            … current_feature_args := get_feature_to_do
            …
            if  current_feature_name.is_equal ("f")
            then
            …
            f_return ?= current_feature_args.item (3)
            f_return_mutex ?= current_feature_args.item (4)
            f_return.copy(f)
            f_return_mutex.unlock
            -- now execution is back at CB at after
            -- the second x_mutex.lock
            …
        …
```

As we can see from the listing above the assignment `x:=y.f` will be translated into y.set_feature_to_do([Current,"f_STRING", x, x_mutex]), where `x` and its associated mutex `x_mutex` are passed as additional arguments (in the case of commands this is not necessary). We thus register the call of feature `f` with subsystem CA. Then we wait to acquire the lock on `x` second time (`x_mutex.lock`). Now we will be waiting until the lock is released in the CA subsystem. In feature `execute` of CA all the references to function *f* are retrieved with the help of the feature `get_feature_to_do`. The reference to `x` is placed into `f_return`, and the reference to `x_mutex` is placed into `f_return_mutex`. Then feature `f` is executed and its result is assigned to `f_return`, which is pointing to the same location in memory as `x`. Mutex `f_return_mutex` can then be released (`f_return_mutex.unlock`) and the execution will continue in subsystem CB, which will get hold of `x_mutex` and will release it. Then the value of `x` can be used for the further calculations.

4.6 One-zero example

Consider the *one-zero* example shown in the BON diagram in Figure 4-7. This example will be used to illustrate the code mappings and the operation of the Generator. There are three classes: `ROOT_CLASS` (shown in Figure 4-8), `PROCESS` (shown in Figure 4-9) and `DATA` (Appendix C).

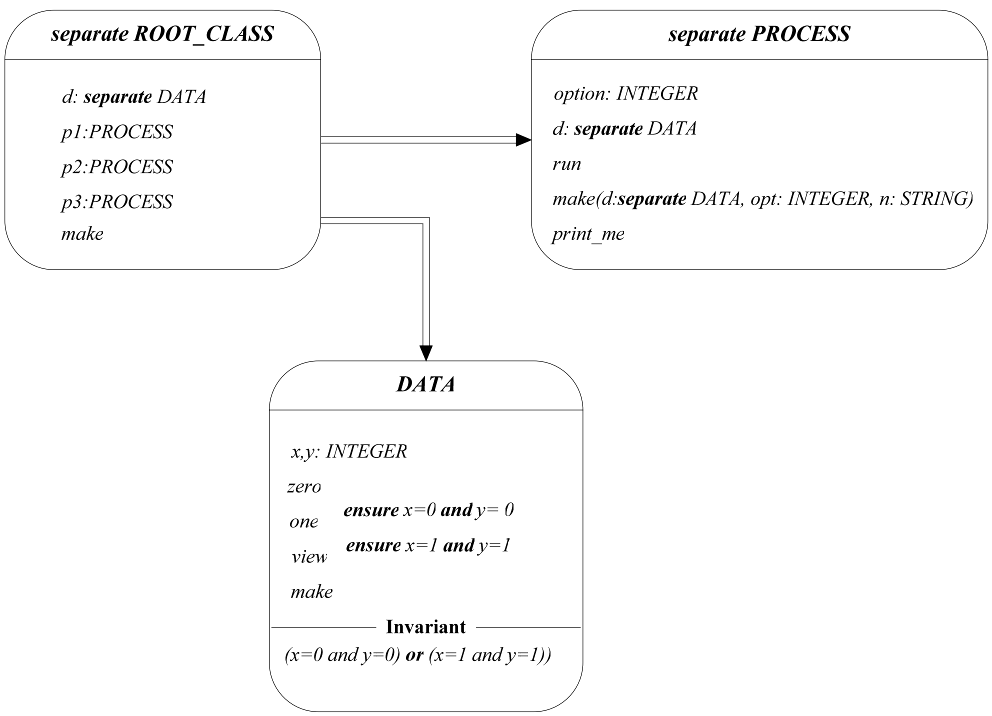

Figure 4-7 BON diagram of zero-one

As can be seen from the BON diagram in Figure 4-7, class DATA has two integer attributes `x` and `y`, and an invariant that asserts that both must either be zero or one. Routine `zero` sets both attributers to zero and routine `one` sets them both to one.

Class PROCESS has access to an instance d of type DATA. Class PROCESS also has an integer `option`, which can be passed via the make routine. An option of 0 tells routine `run` to call `d`'s `zero` routine, an option of 1 tells run to call the `d`'s `one` routine, and an option 2 tells run to call the `d`'s `view` routine.

```
separate class ROOT_CLASS creation
   make
feature
      d: separate DATA
      p1, p2, p3: PROCESS -- separate class

      make is -- start three processes
        do
          io.putstring ("Test threads%N")
          create d.make
          create p1.make(d,0,"First")
          create p2.make(d,1,"Second")
          create p3.make(d,2,"Third")
          p1.run
          p2.run
          p3.run
        end
 end
```

Figure 4-8 ROOT_CLASS for 'One-zero' example

The creation procedure of the root class creates three processes p1, p2 and p3 (of type PROCESS). The same data instance is passed to each process (with options 0, 1 and 2 respectively). Thus all three processes access the data subsystem simultaneously.

In a simple-minded implementation, each process operates in its own thread and the zero and one routines in DATA are protected with a lock local to DATA. Without the mapping of this chapter (each subsystem with its own call buffer etc.), the resulting system will suffer from the concurrent precondition paradox (section 3.7). Thus, there will be an invariant violation in DATA because one process may change the state of x and y after an unlock but before the invariant is evaluated.

```
separate class PROCESS creation
      make
feature
      option: INTEGER
      data: separate DATA
      name: STRING

      run is
         local i:INTEGER
      do
       from until false
       loop
        if option = 0 then
          data.zero -- set data to zero
         elseif option = 1 then
          data.one -- set data to one
        else data.view;
          print_me
        end
       end
      end

      make(d: separate DATA; opt:INTEGER; n:STRING) is
       do
        data := d
        option := opt
        name := n
       end

      print_me is
       do
        print("%N" + name + " just ran" + "%N")
       end
 end
```

Figure 4-9 PROCESS for 'One-zero' example

SCOOP creation routine	Generated Code
separate class	**class** ROOT_CLASS **inherit**
ROOT_CLASS	THREAD
feature	**feature**
	request_pended: INTEGER_REF
	requests_pended_mutex: MUTEX
p1,p2,p3: PROCESS	p1, p2, p3: PROCESS
d: **separate** DATA	d:DATA
	d_mutex: MUTEX
make **is**	make **is**
do	**do**
	requests_pended := 1
	create d_mutex
create d.make	d_mutex.lock
	create d.make
	d_mutex.unlock
create p1.make	**create** p1.make (d, d_mutex, 0,"First",
(d,0,“First”)	requests_pended, requests_pended_mutex)
	p1.launch
create p2.make	**create** p2.make (d, d_mutex, 1, "Second",
(d,0,“Second”)	requests_pended, requests_pended_mutex)
	p2.launch
create p3.make	**create** p3.make (d, d_mutex, 2, "Third",
(d,0,“Third”)	requests_pended, requests_pended_mutex)
	p3.launch
p1.run	p1.set_feature_to_do ([Current,
	"RUN_STRING"])
p2.run	p2.set_feature_to_do ([Current,
	"RUN_STRING"])
p3.run	p3.set_feature_to_do ([Current,
	"RUN_STRING"])
	requests_pended_mutex.lock
	requests_pended.copy(requests_pended-1)
	requests_pended_mutex.unlock
	join_all
end	**end**

Figure 4-10 Mapping from SCOOP to generated code for creation procedure

We now describe how the zero-one example is mapped to generated code. Consider the creation procedure `make` in the `ROOT_CLASS` (Figure 4-8). The mapping from SCOOP to generated code for make is shown in Figure 4-10. Recall that the generated code for the root class must manage the `requests_pended` global variable that keeps track of all `separate` calls across all subsystems (Section 4.2). The mapping works as follows:

1. At lines 1-2 in Figure 4-10, the generated ROOT_CLASS must inherit from THREAD.
2. At lines 4-5 `requests_pended` is declared with its lock.
3. At line 9, the `separate` keyword is stripped from `d`, and a corresponding lock `d_mutex` is declared (section 4.2).
4. At line 14, `requests_pended` is temporarily initialized to a value of 1. As shown in Figure 4-3 (section 4.3) the stopping condition for the `execute` routines in the generated code for the PROCESS subsystems is when `requests_pended` reaches zero. At lines 42-44 `requests_pended` will be decremented by one, but this is only after the top-level calls `p1.run`, `p2.run` and `p3.run` are registered with the appropriate subsystems. Thus, the subsystem `execute` routines are guaranteed not to exit until all the top-level routines (and hence all their sub-calls) are registered and executed. Recall that `requests_pended` is incremented by one at every call registered and decremented by one when the call is later executed (section 4.2).

5. At lines 16-18, the call to create `d` is wrapped with the appropriate lock.

6. At lines 21-23, process `p1` is created. In the generated code, additional arguments must be passed in the creation routine. Data `d` must be passed with its lock, and `requests_pended` with its lock. Then `p1.launch` is called to initiate the thread (and hence the subsystem). In turn, `launch` calls `p1`'s `execute` (effected from `THREAD`) as described in Section 4.2.

7. The same is done for the other two processes `p2` (at lines 25-27) and `p3` (at lines 29-31).

8. Each call to a subsystem must be translated into a register of the calls with the subsystem's buffer (section 4.2). The SCOOP call `p1.run` at line 33 must thus be mapped to a buffer call via `set_features_to_do`. The same call registration must take place for `p2.run` (lines 36-37) and `p3.run` (lines 39-40).

9. At lines 42-44, `requests_pended` must be decremented as explained above in step 4, wrapped with the appropriate lock.

10. At line 47, `join_all` must be invoked. When all `execute` routines in the various subsystems exit (when `requests_pended` reaches zero), then the system exits and terminates.

4.7 Mapping separate preconditions to wait conditions

In section 3.7, we described a `separate` call semantics so that SCOOP `separate` preconditions are treated as wait conditions rather than correctness conditions. We must now show how to map the SCOOP routine shown in Figure 4-11 to generated code.

```
separate class SOME_CLASS ..
      a: separate TYPE

      r(x1: separate TYPE1, x2: separate: TYPE2; x3: TYPE3) is
            require
              x1_validity: x1 /=Void
              x2_validity: x2 /= Void
              a_validity:  a /= Void
              x3_validity: x3 /= Void
            do
              -- routine's body
            end
      end
      …
end
```

Figure 4:11 Separate preconditions (from Section 3.7)

The generated code is shown in Figure 4-12.

```
r(x1: TYPE1; x1_mutex:MUTEX; x2: TYPE2; x2_mutex:MUTEX; x3: TYPE3) is
      require
            x3_validity: x3 /= Void
      local
            scoop_require_wait_flag: BOOLEAN
            access_lock: ACCESS_LOCK
      do
            from
                  scoop_require_wait_flag := False
                  create global_lock
            until
                  scoop_require_wait_flag
            loop
                  global_lock.data.mutex.lock
                  x1_mutex.lock
                  x2_mutex.lock
                  a_mutex.lock
                  global_lock.data.mutex.unlock
                  if  (x1 /=Void) and (x2 /=Void) and (a /=Void)
                              then
                        -- body
                        scoop_require_wait_flag := True
                  end
                  x1_mutex.unlock
                  x2_mutex.unlock
                  a_mutex.unlock
                  sleep(n) --  default n = 50 millisceconds
              end
        end
```

Figure 4.12 Mapping of separate preconditions to wait conditions

The non-separate precondition remains a regular **require** clause as shown in Figure 4-12. A busy-wait loop must be constructed for the `separate` preconditions. In the loop:

- We block until we obtain a lock on all the `separate` entities;
- Once all the `separate` entities are locked, we evaluate all the `separate` preconditions and set an exit flag if they all evaluate to true;
- We unlock all the `separate` entities thus allowing other subsystems to access them;
- Finally we `sleep` for a number of time units that is an option in the Generator before checking the `separate` preconditions again. This ensures that the busy-wait loop does not use up time cycles unnecessarily.

The code

```
global_lock.data.mutex.lock
x1_mutex.lock
x2_mutex.lock
a_mutex.lock
global_lock.data.mutex.unlock
```

makes use of the singleton design pattern to create a global lock. The class GLOBAL_LOCK has a feature *mutex*. The class ACCESS_LOCK has a once routine *data* of type GLOBAL_LOCK. Thus, `global_lock.data.mutex` always refers to the same global mutex. This prevents the type of deadlock in which one process has a handle on `x1_mutex` and another process on `x2_mutex`.

4.8 The Generator

The Generator is invoked as follows:

generator *<input-folder>* *<scoop-project-file-name>* *<output-folder>* [*<sleep>*]

where *sleep* is a nonnegative integer in milliseconds (the sleep parameter in the busy-wait loop). All SCOOP `separate` classes in corresponding **.es* files must be in the *input-folder*. The generated standard Eiffel **.e* files are placed in the *output-folder*. The *scoop-project-file-name* is an Eiffel SCOOP project file (**.esp*) as described in Section 4.1. It is similar to an Eiffel Ace file.

The Generator extracts each class *C* in the project **.esp* file and processes the SCOOP classes one by one. Each class *C* is translated to generated class *CG* and saved as an appropriate text file in the output folder.

Case 1: If C is the root class then the Generator proceeds as follows:

- `CG` inherits from THREAD_CONTROL and uses the mapping in Sections 4.1 and 4.2 for `requests_pended` and the creation routine.
- The Generator scans the rest of the file line by line until the **separate** keyword is found.
- The Generator then uses the appropriate mapping depending on whether the keyword is involved in:
    - a `separate` attribute (section 4.2);
    - a `separate` routine (section 4.5)

Case 2: If C is a `separate` class that is not the root, then

- CG inherits from THREAD.

- The `request_buffer` queue and `execute` routines are inserted into CG as described in sections 4.3 and 4.4.
- The Generator scans the rest of the file line by line until the **separate** keyword is found.
- The Generator then uses the appropriate mapping depending on whether the keyword is involved in:
    - a `separate` attribute (section 4.2);
    - a `separate` routine (section 4.5)

The Generator's accepting grammar is a subset of the Eiffel grammar. It has the following restrictions:

- Each command must be on a `separate` line;
- Consequently the use of ';' to `separate` commands on one line is unsupported;
- The keywords must be lower case and the creation clauses are denoted only by the keyword `creation`.
- Other than the `separate` keyword, Generator assumes that we are dealing with legal Eiffel text.
- The `separate` keyword is illegal as a local entity of a routine. Since local entities can only be accessed by the encapsulating feature clause, it would be nonsensical to declare a local entity as `separate` due to the guarantee that only one processor is allocated per object, and therefore there will only be one thread at the feature level handling it.

- Only one instance of the root class is allowed, as this class manages `requests_pended`.

The original version of the Generator was developed using ISE EiffelStudio 5.0. It has remained compatible up to an including the current version 5.4. While the ISE Eiffel compiler was used to translate the generated code from Eiffel to C, the Microsoft C compiler (included with Visual Studio .NET) was used to compile from C to executable code. Due to major differences in C to executable code compilation, the Borland C compiler generates invalid code when compiling the generator. At the time of this writing, ISE is investigating the problem.

The Generator was tested on a number of examples of different complexity. The target code produced by Generator was compiled on different platforms (Windows, Unix, Linux, Mac) using standard Eiffel compilers.

Chapter 5 – Conclusion

Eiffel's powerful features such as Design by Contract, genericity, multiple inheritance, and seamless and reversible design and code generation via BON, make it a productive environment for developing quality code.

The SCOOP framework adds concurrency to Eiffel, via the addition of only one keyword (**separate**), while preserving all the other features of Eiffel. This concurrent framework removes many areas of difficulty in concurrent programming. In particular, constructs involving mutual exclusion, atomicity, condition variables and synchronization are considerably simplified, and issues such as the inheritance anomaly are virtually removed.

Until this thesis, the only SCOOP implementation was that of (Compton 2000). The Compton implementation was a groundbreaking work implementing many of the features of SCOOP for the first time. The Compton implementation did not however implement some main features of SCOOP. For example, the conversion of `separate` preconditions into wait conditions was not supported. Nor were "once" globals correctly implemented. Also, Compton's implementation was via a compiler modification to an open source compiler called SmallEiffel. However, Compton's work is no longer compatible with the latest version of this compiler (called SmartEiffel).

In this thesis we followed a different approach. Instead of modifying a compiler, we describe and build a Generator that automatically maps SCOOP programs to standard Eiffel together with a cross-platform thread library (Eiffel+Threads). This approach allows us to study SCOOP while maintaining compatibility with new compiler developments. The downside is that we do not have a SCOOP debugger but must use the standard Eiffel debuggers instead. The main contributions of this thesis include:

- An analysis of Meyer's SCOOP framework especially the various implementation issues that arise in this context. In chapter 3, we develop a model of SCOOP using the notion of a subsystem.
- In Chapter 4 we provide a mapping from SCOOP programs to code in Eiffel+Threads in terms of the model;
- Chapter 4 also describes a Generator, implemented in standard Eiffel that automatically translates SCOOP code to executable multi-threaded Eiffel using the mapping.
- We thus provide the first workable and complete cross-platform SCOOP capability that should prove easy to maintain even in the face of new Eiffel compiler enhancements.
- This SCOOP implementation is fully compatible with all standard Eiffel features such as DbC, genericity and multiple inheritance.

The SCOOP Generator should be seen as a ("proof of concept") prototype at this point rather than industrial strength, until such time as its efficiency and correctness has been validated against many large examples.

5.1 Future work

Some design decisions in the current Generator may need to be re-evaluated.

- In section 4.2, the decision was made to implement `separate` attributes as conceptual subsystems rather than as actual subsystems. Further study of the whole issue is needed as indicated in Section 4.2.
- In section 4.5.2, the busy-wait loop for converting `separate` preconditions into wait conditions used a sleep mechanism so as not to waste CPU cycles. A solution using condition variables might prove to be more efficient.

5.2 Model Driven Development

As (Selic 2003) points out, using models to design complex systems is an important part of traditional development. Models help us understand a complex problem and its potential solutions through abstraction. Therefore, it seems obvious that software systems, which are often among the most complex engineering systems, can benefit greatly from using models and modelling techniques.

Surprisingly models in software engineering are used infrequently and, even when used, they often play a secondary role. Yet, as (Selic 2003) writes, the potential benefits of using models are significantly greater in software than in any other engineering discipline. Model-driven development (MDD) methods were devised to take advantage of this opportunity, and the claim is now being made that accompanying technologies have matured to the point where they are generally useful (Mellor 2002).

UML version 2.0 has been developed with MDD in mind. The major advantage of the model-enhanced UML is that we express models using concepts that are much less bound to the underlying implementation technology and are much closer to the problem domain relative to most popular programming languages. This level of abstraction makes the models easier to specify, understand, and maintain.

The UML notion of structured classes and components having their own thread of execution is considered to be an important building block of MDD, together with traditional models such as class diagrams, statecharts and collaboration diagrams. A `separate` SCOOP class would appear to be an abstract version of the UML notion of a structured class. For example, if we want to create a diagram of Producer-Consumer in Java, we will need to go quite low-level to show inheritance from *THREAD, synchronized methods, wait-notifyAll* (Figure 5-2). To present the same in SCOOP we will just have to declare CONSUMER and PRODUCER as `separate` in our diagram (Figure 5-1). This brings us into totally different level of abstraction, allowing us to model software on much higher level. Thus SCOOP may have an important role to play in the currently evolving MDD frameworks.

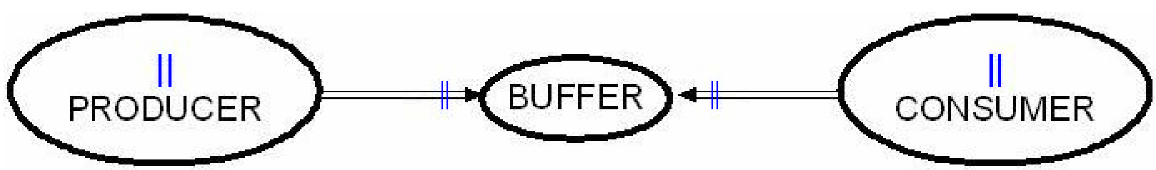

Figure 5-1 Producer-Consumer SCOOP

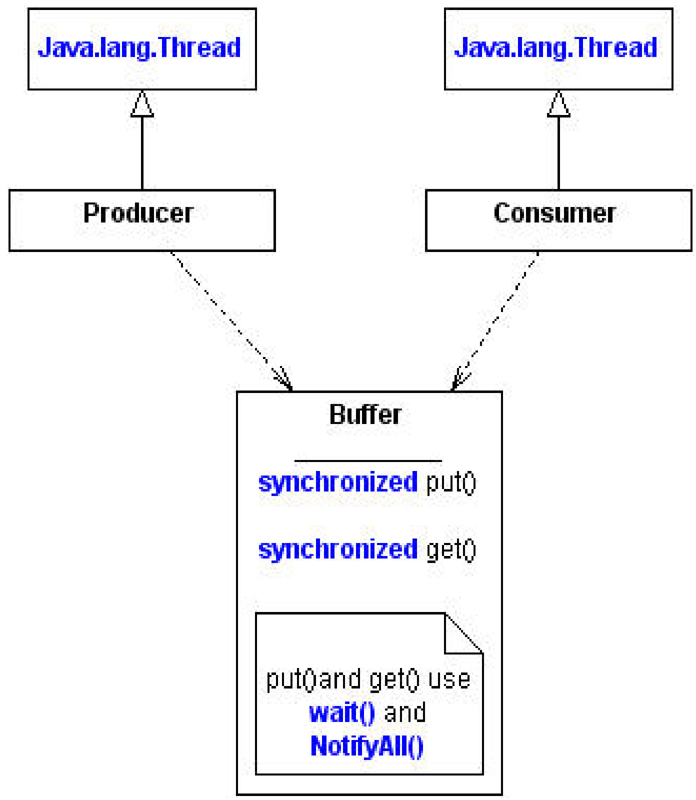

Figure 5-2 Producer-Consumer Java

Appendices

Appendix A. Eiffel Thread Class

```
indexing
        description: "Class defining an Eiffel thread."
        status: "See notice at end of class."
        date: "$Date: 2003/04/25 22:53:21 $"
        revision: "$Revision: 1.1 $"

deferred class
        THREAD

inherit
        THREAD_CONTROL

feature -- Access

        thread_id: POINTER
                        -- Pointer to the thread-id of the current thread object.

feature -- Basic operations

        execute is
                        -- Routine executed by new thread.
                deferred
                end

        launch is
                        -- Initialize a new thread running `execute'.
                do
                        create_thread (Current, $thr_main)
                        thread_id := last_created_thread
                end

        launch_with_attributes (attr: THREAD_ATTRIBUTES) is
                        -- Initialize a new thread running `execute', using attributes.
                do
                        create_thread_with_args (Current, $thr_main, attr.priority,
                                        attr.scheduling_policy, attr.detached)
                        thread_id := last_created_thread
                end

feature {NONE} -- Implementation

        frozen thr_main is
                do
                        thread_id := get_current_id
```

```
                execute
            end

feature {NONE} -- Externals

        create_thread (current_obj: THREAD; init_func: POINTER) is
                -- Initialize and start thread.
            external
                "C signature (EIF_OBJECT, EIF_POINTER) use %"eif_threads.h%""
            alias
                "eif_thr_create"
            end

        create_thread_with_args (current_obj: THREAD; init_func: POINTER; priority, policy:
                                        INTEGER; detach: BOOLEAN) is
                -- Initialize and start thread, after setting its priority
                -- and scheduling policy.
            external
                "C signature (EIF_OBJECT, EIF_POINTER, EIF_INTEGER,
                EIF_INTEGER, EIF_BOOLEAN) use %"eif_threads.h%""
            alias
                "eif_thr_create_with_args"
            end

end -- class THREAD
```

Appendix B. Eiffel Mutex Class

```
indexing
	description: "Mutex synchronization object, allows threads to access global data through critical
sections."
	status: "See notice at end of class."
	date: "$Date: 2003/07/25 20:48:08 $"
	revision: "$Revision: 1.4 $"

class
	MUTEX

inherit
	MEMORY
		redefine
			dispose,
			default_create
		end

create
	default_create,
	make

feature -- Initialization

	default_create is
			-- Create mutex.
		do
			mutex_pointer := eif_thr_mutex_create
		ensure then
			valid_mutex: mutex_pointer /= default_pointer
		end

	make is
		obsolete "Use `default_create'"
			-- Create mutex
		do
			default_create
		ensure
			valid_mutex: mutex_pointer /= default_pointer
		end

feature -- Access

	is_set: BOOLEAN is
			-- Is mutex initialized?
		do
			Result := (mutex_pointer /= default_pointer)
		end

feature -- Status setting

	trylock: BOOLEAN is
```

```
                    -- Has client been successful in locking mutex without waiting?
                    -- Was declared in MUTEX as synonym of `has_locked'.
          require
                    valid_mutex: is_set
          do
                    Result := eif_thr_mutex_trylock (mutex_pointer)
          end

     has_locked: BOOLEAN is
                    -- Has client been successful in locking mutex without waiting?
                    -- Was declared in MUTEX as synonym of `trylock'.
          require
                    valid_mutex: is_set
          do
                    Result := eif_thr_mutex_trylock (mutex_pointer)
          end

     lock is
                    -- Lock mutex, waiting if necessary until that becomes possible.
          require
                    valid_mutex: is_set
          do
                    eif_thr_mutex_lock (mutex_pointer)
          end

     unlock is
                    -- Unlock mutex.
          require
                    valid_mutex: is_set
          do
                    eif_thr_mutex_unlock (mutex_pointer)
          end

     destroy is
                    -- Destroy mutex.
          require
                    valid_mutex: is_set
          do
                    eif_thr_mutex_destroy (mutex_pointer)
                    mutex_pointer := default_pointer
          end

feature {CONDITION_VARIABLE} -- Implementation

     mutex_pointer: POINTER
                    -- C reference to the mutex.

feature {NONE} -- Removal

     dispose is
                    -- Called by the garbage collector when the mutex is
                    -- collected.
          do
                    if is_set then
                              destroy
                    end
```

```
                end

feature {NONE} -- Externals

        eif_thr_mutex_create: POINTER is
                external
                        "C | %"eif_threads.h%""
                end

        eif_thr_mutex_lock (a_mutex_pointer: POINTER) is
                external
                        "C blocking use %"eif_threads.h%""
                end

        eif_thr_mutex_unlock (a_mutex_pointer: POINTER) is
                external
                        "C | %"eif_threads.h%""
                end

        eif_thr_mutex_trylock (a_mutex_pointer: POINTER): BOOLEAN is
                external
                        "C blocking use %"eif_threads.h%""
                end

        eif_thr_mutex_destroy (a_mutex_pointer: POINTER) is
                external
                        "C | %"eif_threads.h%""
                end

end -- class MUTEX
```

Appendix C. One-zero example

Listing 1a. SCOOP `ROOT_CLASS`

```
class
    ROOT_CLASS

create
    make

feature

    d: separate DATA

    p1: PROCESS

    p2: PROCESS

    p3: PROCESS

    make is
        do
            io.putstring ("Test threads%N")
            create d.make
            create p1.make (d, 0, "First")
            create p2.make (d, 1, "Second")
            create p3.make (d, 2, "Third")
            p1.run
            p2.run
            p3.run
        end

end -- class ROOT_CLASS
```

Listing 1b. Generated `ROOT_CLASS`

```
class
        ROOT_CLASS

inherit


        THREAD_CONTROL

create
        make

feature

        requests_pended: INTEGER_REF
                        -- added by generator

        d_mutex: MUTEX
                        -- Added by generator

        requests_pended_mutex: MUTEX
                        -- added by generator

        is_requests_pended: BOOLEAN is
                do
                        Result := True
                        requests_pended_mutex.lock
                        if  requests_pended.is_equal (0) then
                                Result := False
                        end
                        requests_pended_mutex.unlock
                end


        d: DATA

        p1: PROCESS

        p2: PROCESS

        p3: PROCESS

        make is
                do
                        create  requests_pended_mutex.default_create
                        requests_pended := 1
                        create  d_mutex.default_create
                        io.putstring ("Test threads%N")
                        d_mutex.lock
                        create  d.make
                        d_mutex.unlock
```

```
            create  p1.make (d, d_mutex, 0, "First", requests_pended,
                                            requests_pended_mutex)
            p1.launch
            create  p2.make (d, d_mutex, 1, "Second", requests_pended,
                                            requests_pended_mutex)
            p2.launch
            create  p3.make (d, d_mutex, 2, "Third", requests_pended,
                                            requests_pended_mutex)
            p3.launch
            p1.set_feature_to_do ([Current, "RUN_STRING"])
            p2.set_feature_to_do ([Current, "RUN_STRING"])
            p3.set_feature_to_do ([Current, "RUN_STRING"])

                requests_pended_mutex.lock
                requests_pended.copy (requests_pended - 1)
                requests_pended_mutex.unlock
            join_all
        end

end  -- class ROOT_CLASS
```

Listing 2a. SCOOP `PROCESS` Class

```
separate class
        PROCESS

create
        make

feature


        option: INTEGER
                        -- option 0 sets x,y in shared data d to zero
                        -- option 1 sets x,y in shared data d to one
                        -- option 2 just views and prints the shared data


        data: separate DATA -- shared data from calling process

        name: STRING
                        -- name of this process

        make (d: separate DATA; opt: INTEGER; n: STRING) is
                do
                        data := d
                        option := opt
                        name := n
                end

        run is
                        -- option 0 sets x,y in shared data d to zero
                        -- option 1 sets x,y in shared data d to one
                        -- option 2 just views and prints the shared data
                do
                        from
                        until
                                False
                        loop
                                if  option = 0 then
                                        data.zero
                                elseif  option = 1 then
                                        data.one
                                else
                                        data.view
                                        print_me
                                end
                        end
                end
```

```
print_me is
            -- print this process name
      do
            print ("%N" + name + " just ran" + "%N")
      end

end  -- class PROCESS
```

Listing 2b. Generated `PROCESS` Class

```
class
    PROCESS

inherit
    THREAD


create
    make

feature

    execute is
        do
            from
            until
                not is_requests_pended
            loop
                current_feature_args := get_feature_to_do
                current_feature_name ?= current_feature_args.item (2)
                if not current_feature_name.is_equal ("NOTHING") then
                    if current_feature_name.is_equal ("RUN_STRING") then
                        run
                    end
                    if current_feature_name.is_equal ("PRINT_ME_STRING")
                                                                        then
                        print_me
                    end
                    requests_pended_mutex.lock
                    requests_pended.copy (requests_pended - 1)
                    requests_pended_mutex.unlock
                    request_buffer_mutex.lock
                    request_buffer.start
                    request_buffer.remove
                    request_buffer_mutex.unlock
                end
            end
        end

    data_mutex: MUTEX -- Added by generator

     -- Added by generator

    requests_pended: INTEGER_REF

    requests_pended_mutex: MUTEX

    request_buffer: LINKED_LIST [TUPLE]

    request_buffer_mutex: MUTEX
```

```
current_feature_args: TUPLE

current_feature_name: STRING

is_requests_pended: BOOLEAN is
        do
                Result := True
                requests_pended_mutex.lock
                if  requests_pended.is_equal (0) then
                        Result := False
                end
                requests_pended_mutex.unlock
        end

set_feature_to_do (feature_params_arg: TUPLE) is
        do
                requests_pended_mutex.lock
                requests_pended.copy (requests_pended + 1)
                requests_pended_mutex.unlock
                request_buffer_mutex.lock
                request_buffer.extend (feature_params_arg)
                request_buffer_mutex.unlock
        end

get_feature_to_do: TUPLE is
        do
                request_buffer_mutex.lock
                if  not  request_buffer.is_empty then
                        Result := request_buffer.first
                else
                        Result := [Current, "NOTHING"]
                end
                request_buffer_mutex.unlock
        end


option: INTEGER
                -- option 0 sets x,y in shared data d to zero
                -- option 1 sets x,y in shared data d to one
                -- option 2 just views and prints the shared data

data: DATA
                -- shared data from calling process

name: STRING
                -- name of this process



make (d: DATA; d_mutex: MUTEX; opt: INTEGER; n: STRING; requests_pended_arg: INTEGER_REF;
                                                            requests_pended_mutex_arg: MUTEX) is
        do
                requests_pended := requests_pended_arg
                requests_pended_mutex := requests_pended_mutex_arg
                current_feature_name := "NOTHING"
```

```
                create  current_feature_args.make
                create  request_buffer.make
                create  request_buffer_mutex.default_create
                create  data_mutex.default_create
                data_mutex := d_mutex
                data := d
                option := opt
                name := n
            end

    run is
                -- option 0 sets x,y in shared data d to zero
                -- option 1 sets x,y in shared data d to one
                -- option 2 just views and prints the shared data
            do
                from
                until
                    False
                loop
                    if  option = 0 then
                        data_mutex.lock
                        data.zero
                        data_mutex.unlock
                    elseif  option = 1 then
                        data_mutex.lock;
                        data.one;
                        data_mutex.unlock
                    else
                        data_mutex.lock
                        data.view
                        data_mutex.unlock
                        print_me
                    end
                end
            end

    print_me is
                -- print this process name
            do
                print ("%N" + name + " just ran" + "%N")
            end

end  -- class PROCESS
```

Listing 3. `DATA` class

```
indexing
        description: "data to illustrate pthreads"
        author: "JSO"
        date: "$Date: $"
        revision: "$Revision: $"

class
        DATA

inherit
        ANY

create
        make

feature

        x: INTEGER
                        -- Was declared in DATA as synonym of `y'.

        y: INTEGER
                        -- Was declared in DATA as synonym of `x'.

        make is
                        -- set to zero
                do
                        x := 0
                        y := 0
                end

        zero is
                do
                        x := 0
                        y := 0
                end

        one is
                do
                        x := 1
                        y := 1
                end

        get_x: INTEGER is
                        -- gets value of x
                do
                        Result := x
                end

        view is
                do
                        io.put_string ("%NPrinting data x, y%N")
                        io.put_integer (x)
                        io.put_string ("%T")
```

```
            io.put_integer (y)
            check
                date_view_check: (x = 1 and y = 1) or (x = 0 and y = 0)
            end
        end

end -- class DATA
```

Listing 4. `ONE-ZERO` class diagram

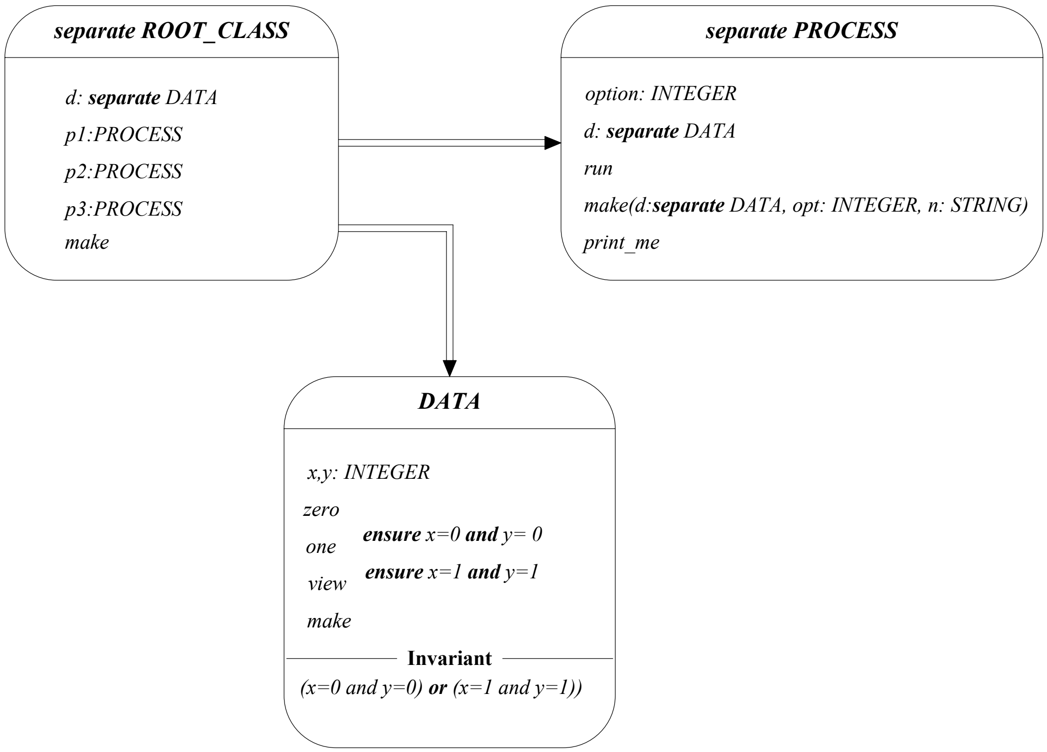

Figure C-1: `ONE-ZERO` class diagram

Appendix D. Consumer – Producer Examples

Listing 1a. SCOOP ROOT_CLASS class

```
class
    ROOT_CLASS

create
    make

feature -- Initialization

    b: separate BUFFER

    p: PRODUCER

    c: CONSUMER

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

end -- class ROOT_CLASS
```

Listing 1b. Generated `ROOT_CLASS` class

```
class
	ROOT_CLASS

inherit
	EXCEPTIONS

	THREAD_CONTROL

create
	make

feature

	requests_pended: INTEGER_REF
			-- added by generator

	b_mutex: MUTEX
			-- Added by generator

	requests_pended_mutex: MUTEX
			-- added by generator

	is_requests_pended: BOOLEAN is
		do
			Result := True
			requests_pended_mutex.lock
			if  requests_pended.is_equal (0) then
				Result := False
			end
			requests_pended_mutex.unlock
		end

	rescue_scoop (who_caused: STRING; what_caused: STRING) is
		do
			io.put_string ("Assertion violated in " + who_caused + ": " + what_caused)
			raise ("Assertion " + what_caused + " violated in " + who_caused)
		end

	b: BUFFER

	p: PRODUCER

	c: CONSUMER

	make is
			-- Creation procedure.
		do
			create  requests_pended_mutex.default_create
			requests_pended := 1
			create  b_mutex.default_create
			b_mutex.lock
			create  b.make
```

```
                    b_mutex.unlock
                    create p.make (b, b_mutex, requests_pended, requests_pended_mutex)
                    p.launch
                    create c.make (b, b_mutex, requests_pended, requests_pended_mutex)
                    c.launch
                    from
                          requests_pended_mutex.lock
                          requests_pended.copy (requests_pended - 1)
                          requests_pended_mutex.unlock
                    until
                          not is_requests_pended
                    loop
                    end
                    join_all
              end

end -- class ROOT_CLASS
```

Listing 2a. SCOOP PRODUCER class

```
separate class
    PRODUCER
create
    make

feature {NONE}

    buffer: separate BUFFER


    make (b: separate BUFFER) is
            -- Initialize `Current'.
        do
            buffer := b
            keep_producing
        end

    keep_producing is
        local
            i: INTEGER
        do
            from
            until
                False
            loop
                i := (i + 1) \\ 5
                produce (buffer, i)
            end
        end

    produce (b: BUFFER; i: INTEGER) is
        require
            b.count <= 2
        do
            b.put (i)
        end

end -- class PRODUCER
```

Listing 2b. Generated `PRODUCER` class

```
class
    PRODUCER

inherit
    THREAD

    EXCEPTIONS

create
    make

feature {NONE}

    execute is
        local
            produce_b: BUFFER
            produce_b_mutex: MUTEX
            produce_i: INTEGER
        do
            from
            until
                not is_requests_pended
            loop
                current_feature_args := get_feature_to_do
                current_feature_name ?= current_feature_args.item (2)
                if not current_feature_name.is_equal ("NOTHING") then
                    if current_feature_name.is_equal
                      ("KEEP_PRODUCING_STRING") then
                        keep_producing
                    end
                    if current_feature_name.is_equal ("PRODUCE_STRING")
                      then
                        produce_b ?= current_feature_args.item (3)
                        produce_b_mutex ?= current_feature_args.item (4)
                        produce_i := current_feature_args.integer_item (5)
                        produce (produce_b, produce_b_mutex, produce_i)
                    end
                    requests_pended_mutex.lock
                    requests_pended.copy (requests_pended - 1)
                    requests_pended_mutex.unlock
                    request_buffer_mutex.lock
                    request_buffer.start
                    request_buffer.remove
                    request_buffer_mutex.unlock
                end
            end
        end

    buffer_mutex: MUTEX
            -- Added by generator
            -- Added by generator
```

```
requests_pended: INTEGER_REF

requests_pended_mutex: MUTEX

request_buffer: LINKED_LIST [TUPLE]

request_buffer_mutex: MUTEX

current_feature_args: TUPLE

current_feature_name: STRING

is_requests_pended: BOOLEAN is
        do
                Result := True
                requests_pended_mutex.lock
                if  requests_pended.is_equal (0) then
                        Result := False
                end
                requests_pended_mutex.unlock
        end

set_feature_to_do (feature_params_arg: TUPLE) is
        do
                requests_pended_mutex.lock
                requests_pended.copy (requests_pended + 1)
                requests_pended_mutex.unlock
                request_buffer_mutex.lock
                request_buffer.extend (feature_params_arg)
                request_buffer_mutex.unlock
        end

get_feature_to_do: TUPLE is
        do
                request_buffer_mutex.lock
                if  not  request_buffer.is_empty then
                        Result := request_buffer.first
                else
                        Result := [Current, "NOTHING"]
                end
                request_buffer_mutex.unlock
        end

        buffer: BUFFER

make (b: BUFFER; b_mutex: MUTEX; requests_pended_arg: INTEGER_REF;
requests_pended_mutex_arg: MUTEX) is
                -- Initialize `Current'.
        do
                requests_pended := requests_pended_arg
                requests_pended_mutex := requests_pended_mutex_arg
                current_feature_name := "NOTHING"
                create  current_feature_args.make
                create  request_buffer.make
                create  request_buffer_mutex.default_create
                create  buffer_mutex.default_create
```

```
            buffer_mutex := b_mutex
            buffer := b
            set_feature_to_do ([Current, "KEEP_PRODUCING_STRING"])
        end

    keep_producing is
        local
            i: INTEGER
        do
            from
            until
                False
            loop
                i := (i + 1) \\ 5
                produce (buffer, buffer_mutex, i)
            end
        end

    produce (b: BUFFER; b_mutex: MUTEX; i: INTEGER) is
        local
            scoop_require_wait_flag: BOOLEAN
        do
            from
                scoop_require_wait_flag := False
            until
                scoop_require_wait_flag
            loop
                b_mutex.lock
                if  (b.count <= 2) then
                    b.put (i)
                    scoop_require_wait_flag := True
                end
                b_mutex.unlock
            end
        end

end  -- class PRODUCER
```

Listing 3a. SCOOP `CONSUMER` class

```
separate class
	CONSUMER

create
	make

feature {NONE}

	buffer: separate BUFFER

	make (b: separate BUFFER) is
			-- Initialize `Current'.
		do
			buffer := b
			keep_consuming
		end

	keep_consuming is
		do
			from
			until
				False
			loop
				consume (buffer)
			end
		end

	consume (b: separate BUFFER) is
		require
			b.count > 0
		do
			b.remove
		end

end -- class CONSUMER
```

Listing 3b. Generated `CONSUMER` class

```
class
    CONSUMER

inherit
    THREAD

    EXCEPTIONS

create
    make

feature {NONE}

    execute is
        local
            consume_b: BUFFER
            consume_b_mutex: MUTEX
        do
            from
            until
                not is_requests_pended
            loop
                current_feature_args := get_feature_to_do
                current_feature_name ?= current_feature_args.item (2)
                if not current_feature_name.is_equal ("NOTHING") then
                    if current_feature_name.is_equal
                       ("KEEP_CONSUMING_STRING") then
                            keep_consuming
                    end
                    if current_feature_name.is_equal ("CONSUME_STRING")
                        then
                            consume_b ?= current_feature_args.item (3)
                            consume_b_mutex ?= current_feature_args.item (4)
                            consume (consume_b, consume_b_mutex)
                    end
                    requests_pended_mutex.lock
                    requests_pended.copy (requests_pended - 1)
                    requests_pended_mutex.unlock
                    request_buffer_mutex.lock
                    request_buffer.start
                    request_buffer.remove
                    request_buffer_mutex.unlock
                end
            end
        end

    buffer_mutex: MUTEX
            -- Added by generator
            -- Added by generator

    requests_pended: INTEGER_REF
```

```
        requests_pended_mutex: MUTEX

        request_buffer: LINKED_LIST [TUPLE]

        request_buffer_mutex: MUTEX

        current_feature_args: TUPLE

        current_feature_name: STRING

        is_requests_pended: BOOLEAN is
                do
                        Result := True
                        requests_pended_mutex.lock
                        if  requests_pended.is_equal (0) then
                                Result := False
                        end
                        requests_pended_mutex.unlock
                end

        set_feature_to_do (feature_params_arg: TUPLE) is
                do
                        requests_pended_mutex.lock
                        requests_pended.copy (requests_pended + 1)
                        requests_pended_mutex.unlock
                        request_buffer_mutex.lock
                        request_buffer.extend (feature_params_arg)
                        request_buffer_mutex.unlock
                end

        get_feature_to_do: TUPLE is
                do
                        request_buffer_mutex.lock
                        if  not  request_buffer.is_empty then
                                Result := request_buffer.first
                        else
                                Result := [Current, "NOTHING"]
                        end
                        request_buffer_mutex.unlock
                end


        buffer: BUFFER


make (b: BUFFER; b_mutex: MUTEX; requests_pended_arg: INTEGER_REF;
                                requests_pended_mutex_arg: MUTEX) is
                        -- Initialize `Current'.
                do
                        requests_pended := requests_pended_arg
                        requests_pended_mutex := requests_pended_mutex_arg
                        current_feature_name := "NOTHING"
                        create  current_feature_args.make
                        create  request_buffer.make
                        create  request_buffer_mutex.default_create
                        create  buffer_mutex.default_create
```

```
            buffer_mutex := b_mutex
            buffer := b
            set_feature_to_do ([Current, "KEEP_CONSUMING_STRING"])
        end

    keep_consuming is
        do
            from
            until
                False
            loop
                consume (buffer, buffer_mutex)
            end
        end

    consume (b: BUFFER; b_mutex: MUTEX) is
        local
            scoop_require_wait_flag: BOOLEAN
        do
            from
                scoop_require_wait_flag := False
            until
                scoop_require_wait_flag
            loop
                b_mutex.lock
                if  (b.count > 0) then
                    b.remove
                    scoop_require_wait_flag := True
                end
                b_mutex.unlock
            end
        end

end  -- class CONSUMER
```

Listing 4. `BUFFER` class

```
class
        BUFFER

create
        make

feature

        count: INTEGER is
                do
                        Result := q.count
                end

        item: INTEGER is
                        -- front
                do
                        Result := q.item
                end

        put (x: INTEGER) is
                        -- enqueue `x'
                require
                        count <= 3
                do
                        q.put (x)
                        print ("PUT")
                        io.new_line
                ensure
                        count = old  count + 1
                        q.has (x)
                end

        remove is
                        -- dequeue
                require
                        count > 0
                do
                        q.remove
                        print ("REMOVE")
                        io.new_line
                ensure
                        count = old  count - 1
                end
```

```
feature {NONE}

    q: QUEUE [INTEGER]

    make is
                -- initialize buffer
        do
                create {ARRAYED_QUEUE [INTEGER]} q.make (3)
        end

invariant

    inv: count <= 3

end -- class BUFFER
```

Listing 5. `PRODUCER-CONSUMER` class diagram

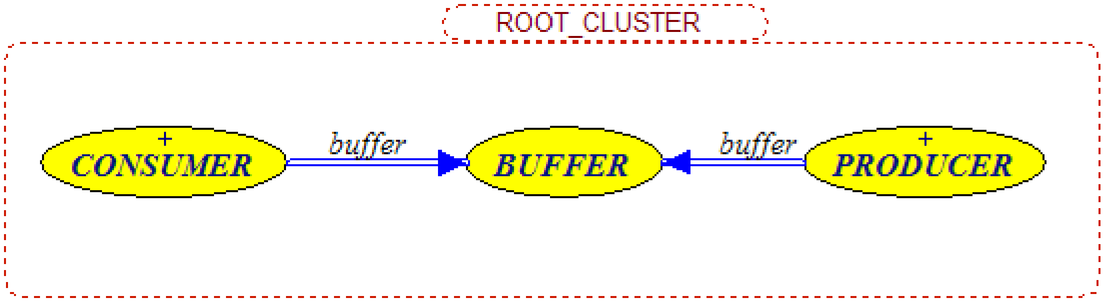

Figure D-1: `PRODUCER-CONSUMER` class diagram

Listing 6a. Java main Producer-Consumer Class

```
public class ProducerConsumer
{
  public static void main(String [] args)
  {
    Buffer b = new Buffer(4);
    Producer p = new Producer(b);
    Consumer c = new Consumer(b);

    p.start();
    c.start();
  }
}
```

Listing 6b. Java Buffer Class

```
public class Buffer {
    protected Object[] buf;
    protected int MAX;
    protected int current = 0;

    Buffer(int max) {
                MAX = max;
                buf = new Object[MAX];
                }

    public synchronized Object get()
                    throws Exception {
                        while (current<=0) { wait(); }
                        current--;
                        Object ret = buf[current];
                        notifyAll();
                        return ret;
                        }

    public synchronized void put(Object v)
                    throws Exception {
                        while (current>=MAX) { wait(); }
                        buf[current] = v;
                        current++;
                        notifyAll();
                        }
}
```

Listing 6c. Java Producer Class

```
class Producer extends Thread {
  private Buffer buffer;

  Producer(Buffer b) { buffer = b; }
  public void run() {
   for(int i = 0; ; (i+1)%5) {
      buffer.Put(i); }
  }
}
```

Listing 6d. Java Consumer Class

```
class Consumer extends Thread {
  private Buffer buffer;

  Consumer(Buffer b) { buffer = b; }
  public void run() {
   for(int i = 0; ; i++) {
      buffer.Get(); }
  }
}
```

Appendix E. THREAD_CONTROL Class

```
indexing
        description: "Control over thread execution."
        status: "See notice at end of class."
        date: "$Date: 2003/07/25 20:48:08 $"
        revision: "$Revision: 1.2 $"

class
        THREAD_CONTROL

feature -- Basic operations

        yield is
                        -- The calling thread yields its execution in favor of another
                        -- thread.
                external
                        "C | %"eif_threads.h%""
                alias
                        "eif_thr_yield"
                end

        join_all is
                        -- The calling thread waits for all other threads to terminate.
                external
                        "C blocking use %"eif_threads.h%""
                alias
                        "eif_thr_join_all"
                end

        join is
                        -- The calling thread waits for the current child thread to
                        -- terminate.
                do
                        thread_wait (Current)
                end

        native_join (term: POINTER) is
                        -- Same as `join' except that the low-level architecture-dependant
                        -- routine is used. The thread must not be created detached.
                do
                        thread_join (term)
                end

feature {NONE} -- Implementation

        terminated: BOOLEAN
                        -- True if the thread has terminated.

feature {NONE} -- Externals

        thread_wait (term: THREAD_CONTROL) is
                        -- The calling C thread waits for the current Eiffel thread to
                        -- terminate.
```

```
        external
            "C blocking use %"eif_threads.h%""
        alias
            "eif_thr_wait"
        end

    thread_join (term: POINTER) is
            -- The calling thread uses the low-level join routine to
            -- join the current Eiffel thread.
        external
            "C blocking use %"eif_threads.h%""
        alias
            "eif_thr_join"
        end

    get_current_id: POINTER is
            -- Returns a pointer to the thread-id of the thread.
        external
            "C | %"eif_threads.h%""
        alias
            "eif_thr_thread_id"
        end

    last_created_thread: POINTER is
            -- Returns a pointer to the thread-id of the last created thread.
        external
            "C | %"eif_threads.h%""
        alias
            "eif_thr_last_thread"
        end

    exit is
            -- Exit calling thread. Must be called from the thread itself.
        external
            "C | %"eif_threads.h%""
        alias
            "eif_thr_exit"
        end

end -- class THREAD_CONTROL
```

Appendix F. Demo_Process Examples

Listing 1a. SCOOP ROOT_CLASS class

```
separate class
    ROOT_CLASS

create
    make

feature
    d:  separate DATA
    p: separate PROCESS


    demo_process: DEMO_PROCESS

    make is
        do
            create  d.make
            create  p.make (d)
            p.some_feature (d)
            create  demo_process.make (p)
            demo_process.demo_feature (d)
        end

end  -- class ROOT_CLASS
```

Listing 1b. Generated `ROOT_CLASS` class

```
class
    ROOT_CLASS

inherit

    THREAD_CONTROL

create
    make

feature

    requests_pended: INTEGER_REF
            -- added by generator

    d_mutex: MUTEX
            -- Added by generator

    p_mutex: MUTEX
            -- Added by generator

    requests_pended_mutex: MUTEX
            -- added by generator

    is_requests_pended: BOOLEAN is
        do
            Result := True
            requests_pended_mutex.lock
            if requests_pended.is_equal (0) then
                Result := False
            end
            requests_pended_mutex.unlock
        end

    d: DATA

    p: PROCESS

    demo_process: DEMO_PROCESS
            --      int_result: INTEGER

    make is
        do
            create requests_pended_mutex.default_create
            requests_pended := 1
            create d_mutex.default_create
            create p_mutex.default_create
            d_mutex.lock
            create d.make
            d_mutex.unlock
            p_mutex.lock
            create p.make (d, d_mutex, requests_pended, requests_pended_mutex)
```

```
                p_mutex.unlock
                p_mutex.lock
                p.launch
                p_mutex.unlock
                p_mutex.lock
                p.set_feature_to_do ([Current, "SOME_FEATURE_STRING", d, d_mutex])
                p_mutex.unlock
                create  demo_process.make (p, p_mutex, requests_pended,
requests_pended_mutex)
                demo_process.launch
                demo_process.set_feature_to_do ([Current, "DEMO_FEATURE_STRING", d,
d_mutex])
                requests_pended_mutex.lock
                requests_pended.copy (requests_pended - 1)
                requests_pended_mutex.unlock
                join_all
            end

end  -- class ROOT_CLASS
```

Listing 2a. SCOOP `PROCESS` class

```
separate class
    PROCESS

create
    make,
    second_make

feature

    data: separate DATA

    make(d:separate DATA) is

        do
            data := d
            data.one
        end

    second_make is
        do
        end

    no_arg_no_res_feature is
        do
        end

    some_feature (d: DATA) is
        require
            d_not_void: d /= void
            d_equal_to_zero: d.x = 0 and  d.y = 0
        do
            d.one
            io.put_integer (d.x)
            io.put_integer (d.y)
        ensure
            d_equal_to_one: d.x = 1 and  d.y = 1
        end

    another_feature: INTEGER is
        do
            Result := data.x
        end

    third_feature (d: DATA; i: INTEGER) is
        require
            d_equal_to_one: d.x = 1 and  d.y = 1
            i_not_zero: i /= 0
        do
            check
                d_not_void: d /= void
            end
            d.zero
```

```
            io.put_integer (d.x)
            check
                d_not_void: d /= void
            end
            io.put_integer (d.y)
        ensure
            d_equal_to_zero: d.x = 0 and  d.y = 0
        end

end  -- class PROCESS
```

Listing 2b. Generated PROCESS class

```
class
    PROCESS

inherit
    THREAD

create
    make,
    second_make

feature

    data: DATA

    execute is
        local
            some_feature_d: DATA
            some_feature_d_mutex: MUTEX
            third_feature_d: DATA
            third_feature_d_mutex: MUTEX
            third_feature_i: INTEGER
        do
            from
            until
                not is_requests_pended
            loop
                current_feature_args := get_feature_to_do
                current_feature_name ?= current_feature_args.item (2)
                if not current_feature_name.is_equal ("NOTHING") then
                    if current_feature_name.is_equal
                      ("NO_ARG_NO_RES_FEATURE_STRING") then
                            no_arg_no_res_feature
                    end
                    if current_feature_name.is_equal
                      ("SOME_FEATURE_STRING") then
                        some_feature_d ?= current_feature_args.item (3)
                        some_feature_d_mutex ?= current_feature_args.item (4)
                        some_feature (some_feature_d, some_feature_d_mutex)
                    end
                    if current_feature_name.is_equal
                        ("THIRD_FEATURE_STRING") then
                        third_feature_d ?= current_feature_args.item (3)
                        third_feature_d_mutex ?= current_feature_args.item (4)
                        third_feature_i := current_feature_args.integer_item (5)
                        third_feature (third_feature_d, third_feature_d_mutex,
                                                            third_feature_i)
                    end
                    requests_pended_mutex.lock
                    requests_pended.copy (requests_pended - 1)
                    requests_pended_mutex.unlock
                    request_buffer_mutex.lock
                    request_buffer.start
```

```
                        request_buffer.remove
                        request_buffer_mutex.unlock
                    end
                end
            end

        data_mutex: MUTEX
                -- Added by generator
                -- Added by generator

        requests_pended: INTEGER_REF

        requests_pended_mutex: MUTEX

        request_buffer: LINKED_LIST [TUPLE]

        request_buffer_mutex: MUTEX

        current_feature_args: TUPLE

        current_feature_name: STRING

        is_requests_pended: BOOLEAN is
            do
                Result := True
                requests_pended_mutex.lock
                if  requests_pended.is_equal (0) then
                    Result := False
                end
                requests_pended_mutex.unlock
            end

        set_feature_to_do (feature_params_arg: TUPLE) is
            do
                requests_pended_mutex.lock
                requests_pended.copy (requests_pended + 1)
                requests_pended_mutex.unlock
                request_buffer_mutex.lock
                request_buffer.extend (feature_params_arg)
                request_buffer_mutex.unlock
            end

        get_feature_to_do: TUPLE is
            do
                request_buffer_mutex.lock
                if  not  request_buffer.is_empty then
                    Result := request_buffer.first
                else
                    Result := [Current, "NOTHING"]
                end
                request_buffer_mutex.unlock
            end

        make (d: DATA; d_mutex: MUTEX; requests_pended_arg: INTEGER_REF;
    requests_pended_mutex_arg: MUTEX) is
            do
```

```
            requests_pended := requests_pended_arg
            requests_pended_mutex := requests_pended_mutex_arg
            current_feature_name := "NOTHING"
            create current_feature_args.make
            create request_buffer.make
            create request_buffer_mutex.default_create
            create data_mutex.default_create
            data_mutex := d_mutex
            data := d
            data_mutex.lock
            data.one
            data_mutex.unlock
        end

    second_make (requests_pended_arg: INTEGER_REF; requests_pended_mutex_arg: MUTEX) is
        do
            requests_pended := requests_pended_arg
            requests_pended_mutex := requests_pended_mutex_arg
            current_feature_name := "NOTHING"
            create current_feature_args.make
            create request_buffer.make
            create request_buffer_mutex.default_create
            create data_mutex.default_create
        end

    no_arg_no_res_feature is
        do
        end

    some_feature (d: DATA; d_mutex: MUTEX) is
        local
            scoop_require_wait_flag: BOOLEAN
        do
            from
                scoop_require_wait_flag := False
            until
                scoop_require_wait_flag
            loop
                d_mutex.lock
                if (d /= void) then
                    if (d.x = 0 and d.y = 0) then
                        d.one
                        io.put_integer (d.x)
                        io.put_integer (d.y)
                        scoop_require_wait_flag := True
                    end
                end
                d_mutex.unlock
            end
        end

    another_feature: INTEGER is
        do
            data_mutex.lock
            Result := data.x
            data_mutex.unlock
```

```
        end

    third_feature (d: DATA; d_mutex: MUTEX; i: INTEGER) is
        require
            i_not_zero: i /= 0
        local
            scoop_require_wait_flag: BOOLEAN
        do
            from
                scoop_require_wait_flag := False
            until
                scoop_require_wait_flag
            loop
                d_mutex.lock
                if  (d.x = 1 and  d.y = 1) then
                    check
                        d_not_void: d /= void
                    end
                    d.zero
                    io.put_integer (d.x)
                    check
                        d_not_void: d /= void
                    end
                    io.put_integer (d.y)
                    scoop_require_wait_flag := True
                end
                d_mutex.unlock
            end
        end

end  -- class PROCESS
```

Listing 3a. SCOOP `DEMO_PROCESS` class

```
separate class
        DEMO_PROCESS

create
        make

feature --          process_var: separate PROCESS

        process_var: PROCESS

        make (p: separate PROCESS) is
                do
                        process_var := p
                end

        demo_feature (d: separate DATA) is
                                        local
                        i: INTEGER
                do
                        i := 10
                        process_var.third_feature (d, i)
                end

end  -- class DEMO_PROCESS
```

Listing 3b. Generated `DEMO_PROCESS` class

```
class
        DEMO_PROCESS

inherit
        THREAD


create
        make

feature

        process_var: PROCESS

        execute is
                local
                        demo_feature_d: DATA
                        demo_feature_d_mutex: MUTEX
                do
                        from
                        until
                                not  is_requests_pended
                        loop
                                current_feature_args := get_feature_to_do
                                current_feature_name ?= current_feature_args.item (2)
                                if  not  current_feature_name.is_equal ("NOTHING") then
                                        if  current_feature_name.is_equal
                                          ("DEMO_FEATURE_STRING") then
                                           demo_feature_d ?= current_feature_args.item (3)
                                           demo_feature_d_mutex ?= current_feature_args.item (4)
                                           demo_feature (demo_feature_d, demo_feature_d_mutex)
                                        end
                                        requests_pended_mutex.lock
                                        requests_pended.copy (requests_pended - 1)
                                        requests_pended_mutex.unlock
                                        request_buffer_mutex.lock
                                        request_buffer.start
                                        request_buffer.remove
                                        request_buffer_mutex.unlock
                                end
                        end
                end

        process_var_mutex: MUTEX
                        -- Added by generator
                        -- Added by generator

        requests_pended: INTEGER_REF

        requests_pended_mutex: MUTEX

        request_buffer: LINKED_LIST [TUPLE]
```

```
request_buffer_mutex: MUTEX

current_feature_args: TUPLE

current_feature_name: STRING

is_requests_pended: BOOLEAN is
        do
                Result := True
                requests_pended_mutex.lock
                if  requests_pended.is_equal (0) then
                        Result := False
                end
                requests_pended_mutex.unlock
        end

set_feature_to_do (feature_params_arg: TUPLE) is
        do
                requests_pended_mutex.lock
                requests_pended.copy (requests_pended + 1)
                requests_pended_mutex.unlock
                request_buffer_mutex.lock
                request_buffer.extend (feature_params_arg)
                request_buffer_mutex.unlock
        end

get_feature_to_do: TUPLE is
        do
                request_buffer_mutex.lock
                if  not  request_buffer.is_empty then
                        Result := request_buffer.first
                else
                        Result := [Current, "NOTHING"]
                end
                request_buffer_mutex.unlock
        end

make (p: PROCESS; p_mutex: MUTEX; requests_pended_arg: INTEGER_REF;
                                        requests_pended_mutex_arg: MUTEX) is
        do
                requests_pended := requests_pended_arg
                requests_pended_mutex := requests_pended_mutex_arg
                current_feature_name := "NOTHING"
                create  current_feature_args.make
                create  request_buffer.make
                create  request_buffer_mutex.default_create
                create  process_var_mutex.default_create
                process_var_mutex := p_mutex
                process_var := p
        end

demo_feature (d: DATA; d_mutex: MUTEX) is
        local
                i: INTEGER
        do
```

```
            i := 10
            process_var_mutex.lock
            process_var.set_feature_to_do ([Current, "THIRD_FEATURE_STRING", d,
                                                              d_mutex, i])
            process_var_mutex.unlock
        end

end -- class DEMO_PROCESS
```

Appendix G. Inheritance Anomaly

Listing 1. Java Buffer class

```
public class Buffer {

   protected Object[] buf;
   protected int MAX;
   protected int current = 0;

   Buffer(int max) {
         MAX = max;
         buf = new Object[MAX];
   }
   public synchronized Object item()
       throws Exception {
         while (current<=0) { wait(); }
         current--;
         Object ret = buf[current];
         notifyAll();
         return ret;
   }
   public synchronized void put(Object v)
       throws Exception {
         while (current>=MAX) { wait(); }
         buf[current] = v;
         current++;
         notifyAll();
   }
}
```

Listing 2. Java Buffer2 class

```
public class Buffer2 extends Buffer {
  boolean afterGet = false;

  public HistoryBuffer(int max) { super(max);

  public synchronized Object item2()
      throws Exception {
        while ((current<=0)||(afterGet)) {
              wait();
        }
        afterGet = false;
        return super.get();
  }
  public synchronized Object item()
      throws Exception {
        Object o = super.get();
        afterGet = true;
        return o;
  }
  public synchronized void put(Object v)
      throws Exception {
        super.put(v);
        afterGet = false;
  }
}
```

References

Agha, G. (1986). "Actors: A Model of Concurrent Computation in Distributed Systems." Series in Artificial Intelligence, MIT Press, Cambridge, MA.

Arjormandi, E., W. G. O'Farrel, I. Kalas, G. Koblents, F. C. Eigler and G. G. Gao (1995). "ABC++: Concurrency by Inheritance in C++." IBM Systems Journal **34**(1): 120-136.

Briot, J.-P. (1996). An experiment in classification and specialization of synchronisation schemes. Proceedings of the Second International Symposium on Object Technologies for Advanced Software (ISOTAS '96). Computer Science, Springer-Verlag, New York.

Briot, J.-P. (1998). "Concurrency and distribution in object-oriented programming." ACM Computing Surveys **Vol. 30**(3).

Bruno, J. and M. Karaorman (1993). "Introducing concurrency to a sequential language." ACM Computing Surveys **ACM 36.**(9): 103-116.

Caromel, D. (1989). "Service, Asynchrony, and Wait-by-Necessity." Journal of Object-Oriented Programming **2(4):12--18**.

Caromel, D. (1990). "Concurrency and reusability: From sequetial to parallel." Object Oriented Program **3**.

Compton, M. (2000). SCOOP: An Investigation of Concurrency in Eiffel. Department of Computer Science, The Australian National University.

Garbinato, B., R. Guerraoui and K. Mazouni (1994). Distributed programming in GARF. Workshop on Object-Based Distributed Programming, Springer-Verlag.

Gunaseelan, L. and R. LeBlank (1992). Distributed Eiffel: A language for programming multi-granular distributed objects. Proceedings of the Fourth International Conference on Computer Languages. IEEE Computer, Los Alamitos, Calif.

ISE (2003). Eiffel Software - the Home of EiffelStudio and Eiffel ENViSioN.

Jalloul (2000). "Communicating Sequential Systems." Journal of Object-Oriented Programming: 7.

Jezequel, J.-M. (1993). "EPEE: An Eiffel Environment to program distributed-memory parallel computers." Object Oriented Program **6**(2).

Kafura, D. and K. Lee (1990). "ACT++: Building a concurrent C++ with actors." Object Oriented Program **3**(1).

Kiczales, G., D. Riviers and D. Bobrow (1991). "The Art of the Meta-Object Protocol." MIT Press, Cambridge, MA.

Lohr, K.-P. (1993). "Concurrency annotations for reusable software." ACM Computing Surveys **ACM 36**(9): 81-89.

Matsuoka, S. and A. Yonezawa (1993). "Analysis of inheritance anomaly in object-oriented concurent programming languages." MIT Press(Research Directions in Concurrent Object-Oriented Programming).

McAffer, J. (1995). Meta-level Programming with CodA. Proceedings of the European Conference on Object-Oriented Programming (ECOOP '95), Springer-Verlag, New York.

McDowell, E. and D. Helmbold (1989). "Debugging Concurrent Programs." ACM Computing Surveys **21**(4): 593-622.

Mellor, S. J. and M. J. Balcer (2002). Executable UML.

Meyer, B. (1997). Object-Oriented Software Construction, Second Edition, Prentice Hall.

Milicia, G. (2003). "The inheritance anomaly: ten years after." Chi Spaces Technologies ltd.

Nienaltowski, P. (2003). "SCOOPLI: a library for concurrent object-oriented programming on .NET."

P.H.M. America: A. Yonezawa and M. Tokoro (1987). "Pool-T: A parallel object-oriented language." Eds. Computer Systems Series, MIT Press, Cambridge, MA.

Selic, B. (2003). "The Pragmatics of Model-Driven Development." IEEE Software.

Tokoro, M. and Y. Yokote (1987). Expirience and evolution of Concurrent SmallTalk. Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languagesand Applications (OOPSLA '87). ACM SIGPLAN Not. ,22 12.

Tsichritzis, D. and O. Nierstrasz (1995). Concurrency in object-oriented programming languages. Englewood Cliffs, NJ, Prentice Hall International.

Van den Bos, J. and C. Laffra (1991). "Procol - A concurrent object oriented language with protocols, delegation and constraints." Acta Inf **28**: 511-538.

Voss, M. J., I. Park and R. Eigenmann (1999). "On the Machine Independent Target Language for Parallelizing Compilers."

Wegner, P. (1990). "Concept and Paradigms of Object-Oriented Programming." ACM Computing Surveys **1**(1).

Yonezawa, A., J.-P. Briot and E. Shibayama (1986). Object-Oriented Concurrent Programming in ABCL/1. Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications(OOPSLA '86). ACM SIGPLAN Not. 21, 11.