Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance

A thesis submitted to
University of Dublin, Trinity College
for the degree of
Doctor of Philosophy

Ciaran McHale
Distributed Systems Group
Department of Computer Science
Trinity College
Dublin 2
Ireland

http://www.dsg.cs.tcd.ie/

October 1994

Declaration

The work described in this thesis is entirely that of the author and has not been submitted as an exercise for a degree at this or any other university.

Signed:

Ciaran McHale
March 30, 1995

Permission to Lend and/or Copy

I agree that Trinity College Library may lend or copy this thesis upon request.

Signed:

Ciaran McHale
March 30, 1995

Summary of Contribution

This thesis concerns itself with the topic of synchronisation in concurrent, object-oriented languages (COOPLs) and makes contributions in three areas.

The Sos paradigm. This thesis introduces the Sos paradigm for the design of synchronisation mechanisms. This paradigm offers several important benefits. In particular:

- Bloom [Blo79] argues that for a synchronisation mechanism to have good expressive power, it must have access to six different types of information: the name of the invoked operation, parameters, instance variables, the relative arrival time of invocations, the current synchronisation state of the object and information about what operations have executed in the past.

  The Sos paradigm shows that all this information needed for good expressive power comes from just a single source. Thus, by accessing this information in its original, unified form, a Sos-based synchronisation mechanism can have at least as much expressive power as other mechanisms that access the information in a piecemeal fashion.

- Many languages [DDR+91] [And81] [Atk90] [Tho92] [GW91] [Tho94] [Löh93] [LL94] [GC92] [Ber94] [Cou94] permit synchronisation code to access instance variables but do not provide any means to ensure that such variables are not accessed while they are in an inconsistent state.

  The Sos paradigm shows that, contrary to popular belief, a synchronisation mechanism does not need to access the instance variables of an object; synchronisation code can maintain its own variables.

- Many languages [Ada83] [MK87] [LR80] [Sel90] [TS89] [KL89] [Nie87] [TA88] [BI92] permit synchronisation code to be mixed in with sequential code in an attempt to increase the expressive power of the language's synchronisation mechanism.

  In contrast, the synchronisation code of a Sos-based mechanism can be completely separated from sequential code, without any loss of expressive power. This separation is a form of modularity which aids code writing, maintenance and reuse.

We illustrate the above benefits of the Sos paradigm via a sample synchronisation mechanism, Esp, the expressive power of which is shown through numerous examples. In particular, Esp excels at specifying scheduling and liveness constraints.

Many synchronisation mechanisms are *declarative* in nature: programmers simply declare (specify) what synchronisation policy they want and the implementation will be derived automatically from this specification. While declarative mechanisms are elegant and concise, they usually have limited expressive power and hence can specify only a few synchronisation policies. In contrast to declarative mechanisms, some mechanisms are *procedural*. In these, synchronisation policies are implemented via algorithms. Although more verbose and complex to use than declarative mechanisms, procedural mechanisms can generally implement a wider range of policies. A unique benefit of our Esp mechanism is that it is both declarative and procedural at the same time, and offers the best of both worlds: the elegance and conciseness of declarative mechanisms is combined with the ability to implement a wide range of synchronisation policies.

Finally, our prototype implementation of the Esp mechanism shows that the concepts of the Sos paradigm can be easily added to an object-oriented language.

Generic synchronisation policies. Several different synchronisation policies, e.g., mutual exclusion and readers/writer, occur frequently in practice. It is error-prone to have to re-implement such policies time and time again in different classes that use them. It would be preferable to write *generic* versions of such policies that could then be instantiated upon the operations of a class as desired. For example, consider a class that contains three operations, A, B and C, that examine some instance variables, and two other operations, D and E, that update instance variables. A suitable policy for this class might be as follows:

ReadersWriter[{A, B, C}, {D, E}]

This is a readers/writer policy that is instantiated upon two sets of operations. The first, "{A, B, C}", is a set of read-style operations and the second, "{D, E}", a set of write-style operations.

Some languages already provide *limited* support for generic synchronisation policies [Atk90] [LL94] [CL91] [BFS93]. In this thesis, we show that Sos paradigm makes it possible to provide comprehensive language support for generic synchronisation policies. The most obvious benefit of generic synchronisation policies is code reuse. However, we demonstrate that generic synchronisation policies offer additional benefits, including facilitating the optimisation of synchronisation code.

Analysis of the problems with the use of inheritance in COOPLs. A common belief is that there is a conflict between synchronisation and inheritance that can hinder the reuse of code [TS89] [KL89] [Neu91] [GW91] [Tho94] [Löh93] [BBI+] [Mes93] [BFS93] [Mat93] [Cou94] [Ber94]. We show that this perception of the conflict is incorrect; rather, the problem is rooted in the conflicting interaction of two different uses of inheritance. We survey the conflict in a variety of inheritance mechanisms and show that the use of generic synchronisation policies can drastically reduce its harmful effects.

Acknowledgements

It is important to have a good supervisor for one's research. I was lucky: I had two. Furthermore, they were not just good but *excellent* at their job. The first was Bridget Walsh. After she left the department, Seán Baker took over the task of supervision. Thank you to both of them.

Thank you to Alexis Donnelly for his enthusiastic interest in my work and his cheerful willingness to proof-read numerous draft documents.

Thank you to the Distributed Systems Group for patiently funding me in a topic that is somewhat tangential to the main thrust of the group's research. I hope that the fruits of my research, which have taken five years to mature, will be of use to the group.

Thank you to Peter Grogono for providing me with the source code to his Dee system—which was used as the basis for the DESP and GASP prototypes—and for his quick responses to all my email queries.

During the course of my research I had the great pleasure of meeting some fellow researchers at conferences. I especially wish to thank Oscar Nierstrasz, Stephen Goldsack, Colin Atkinson and Christian Neusius for the fruitful conversations I have had with them both in person and via email.

Finally, thank you to friends and family for putting up with me, especially during the last few months when I have been so absorbed in the writing of this thesis.

Contents

Part I

Introduction

Introduction to Part I

Part I is an introduction to this thesis. It contains two chapters.

Chapter 1 provides some background information for readers who are not overly familiar with the area of synchronisation for concurrent, object-oriented languages.

Chapter 2 discusses the aims of this thesis in detail.

Chapter 1

Background Information

This chapter provides some background information for readers who might not be familiar with the area of synchronisation in concurrent, object-oriented languages (COOPLs).

Section 1.1 starts with a brief introduction to object-orientation. Section 1.2 provides a brief introduction to concurrency. Sections 1.3 through to 1.6 provide overviews of some synchronisation mechanisms.

1.1 Object-orientation

As the capabilities of computers have grown, so too have the size and complexity of applications written for them. This has resulted in continual development of techniques to help programmers control the inherent complexity of large software systems. Obvious examples include the development of high-level languages to boost productivity over assembler and the dropping of "goto" statements in favour of structured programming. Another important development is that of abstract data types (ADTs) to encapsulate the implementation of a resource and permit other parts of a program to access the resource only via exported operations (procedures and functions). Not only do ADTs aid in constructing modular programs, they also ensure that parts of a program that access a resource are protected from future possible changes to the resource's implementation.

Object-orientation [Mey88] is a refinement of the concept of ADTs. We use the term *type* to denote the specification aspect of an ADT, e.g., its list of exported operations, and the term *class* to denote its implementation.

One refinement that object-orientation adds to ADTs is the concept of *inheritance* which we now briefly discuss. It is common for the code of several classes to be similar. In such cases, repetition of code can be avoided by defining a class as an incremental modification of other classes. The newly defined class is called a *subclass* of its *parent* classes. A subclass might introduce new operations or selectively re-implement some operations inherited from parent classes. Even if a subclass re-implements an operation, the re-implemented operation can make a "super" call to the corresponding operation in a parent class, thereby incrementally

modifying the operation rather than re-implementing it anew. Languages that restrict a class to have at most one parent class are said to employ *single inheritance*. If a class can have multiple parents then the language is said to permit *multiple inheritance*.

Just as one class may be implemented as an incremental modification of another class, so too may one type (a *subtype*) be defined as an incremental modification of another type (its *parent* type).

It is possible to imagine that several classes may implement one type. However, some languages impose a one-to-one mapping between types and their implementations (classes) and then further simplify this by merging the concepts of type and class. While this union offers simplicity, it also has some drawbacks [Ame87, Sny86].

1.2 Concurrent Programing

There are several benefits of concurrent systems. One is that an application that takes a long time to run may be speeded up by dividing the work of the application into separate processes that can be run concurrently on different CPUs. Even on a single CPU system, a concurrent program might run faster than an equivalent sequential program if the input/output operations of one process overlap with the CPU-intensive executions of another task.

Another potential benefit of concurrency is that some applications are naturally concurrent and cannot be easily written in a sequential language. Ideally, such applications should be easier to write in a concurrent language. However, research into language constructs to aid concurrent programming has yet to mature and it can still be difficult to write concurrent applications. It is widely accepted that the encapsulation and modularity offered by the object-oriented paradigm are useful in controlling the inherent complexity of concurrent programming and this has lead to the development of concurrent, object-oriented programming languages (COOPLs).

Uncontrolled concurrency can be dangerous. For example, if one process examines (reads) an object while it is being updated (written) concurrently by another process then the first process might see the object while it is in a temporarily inconsistent state. Similarly, if two processes try to update the same object concurrently then the overlapping updates might leave the object in an inconsistent state. To guard against such problems, a system must provide a means for different processes to *synchronise* with each other whenever they concurrently access the same object. A *synchronisation mechanism* is a set of language constructs (and/or primitive operations provided by the operating system) that programmers use to ensure the correct synchronisation of processes.

The particular synchronisation "protocol" or "policy" that should be used when accessing an instance of a class can vary from one class to another. For example, all the operations of one class might be write-style operations in which case processes will probably access an instance of that class in mutual exclusion (often abbreviated to "mutex"). If a class contains a mixture of read-style and write-style operations then a "multiple readers/single writer"

(usually abbreviated to "readers/writer") policy might be used.

Generally, the code that implements a synchronisation policy is written inside a class, rather than being spread out among all the clients of that class, which aids modularity [Blo79] and robustness.

We now move on to give an overview of several different synchronisation mechanisms, starting with those mechanisms that are based on synchronisation counters.

1.3 Synchronisation Mechanisms Based on Counters

Synchronisation counters were independently developed by Robert and Verjus [RV77] and Gerber [Ger77]. They are variables of an object that count the total number of invocations for each operation of the object that have actually *arrived* at the object, have *started* execution and have *terminated* execution, etc. Synchronisation counters are quite common and can be found in numerous synchronisation mechanisms [DDR+91] [MWBD91] [CGM91] [And79] [GW91] [Cou94]. Usually, there are five counters for each operation of an object. These are:[1]

arrival(Op):		total number of invocations to execute operation *Op* that have arrived at this object.
wait(Op)	:	current number of invocations that are waiting to execute *Op*.
start(Op)	:	total number of invocations that have started executing *Op*.
exec(Op)	:	current number of invocations that are executing *Op*.
term(Op)	:	total number of invocations that have terminated execution of *Op*.

Only three counters, *arrival*, *start* and *term*, need be maintained since the other two can be expressed in terms of them as follows:

$$wait(\mathrm{Op}) = arrival(\mathrm{Op}) - start(\mathrm{Op})$$
$$exec(\mathrm{Op}) = start(\mathrm{Op}) - term(\mathrm{Op})$$

Synchronisation counters are most often used in *guards*: conditions associated with operations of an object. When an operation is invoked, the invocation is blocked until its guard becomes true. An example to illustrate the usage of guards can be seen in the class in Figure 1.1. This example also introduces the pseudo-code notation that is used throughout most of this thesis. The notation should be intuitive to most readers. Braces ("{...}") are used to denote scope. The constructor of a class is denoted by an operation whose name is identical to the name of the class. A keyword, **synchronisation**, separates the sequential operations of a class from their guards. Guards themselves are written in the form:

operation-name: *condition*

[1] Unfortunately, there are no standard names for these counters since different synchronisation mechanisms call them by different names. These are the counter names we use throughout this thesis.

The guards in Figure 1.1 implement a readers/writer policy. An invocation of *Read* is permitted to execute if there are currently no executions of *Write*; similarly, an invocation of *Write* can execute if there are currently no executions of either *Read* or *Write*.

```
class ReadersWriter {
  ...   // declaration of instance variables
  ReadersWriter(...) { ... }   // constructor
  Read(...) { ... }
  Write(...) { ... }
synchronisation
  Read: exec(Write) = 0;
  Write: exec(Read) = 0 and exec(Write) = 0;
}
```

Figure 1.1: A class containing *Read* and *Write* operations and appropriate guards

A shorthand used throughout this thesis is to use "$exec(\mathrm{Op}_1, \mathrm{Op}_2, \ldots, \mathrm{Op}_n) = 0$" instead of the more verbose:

$$exec(\mathrm{Op}_1) = 0 \textbf{ and } exec(\mathrm{Op}_2) = 0 \textbf{ and } \ldots \ exec(\mathrm{Op}_n) = 0$$

Thus, the guard for *Write* in Figure 1.1 could be rewritten as:

Write: $exec$(Read, Write) = 0;

The *wait* counter is often used to implement a priority scheme. For example, a readers priority version of the readers/writer policy can be expressed using the following guards:

Read: $exec$(Write) = 0;
Write: $exec$(Read, Write) = 0 **and** $wait$(Read) = 0;

1.3.1 Guide

Synchronisation counters are used in the guard-based synchronisation mechanism of the Guide language [DDR+91]. The guards can contain not only synchronisation counters but also instance variables of the object and parameters of the operation being invoked.

The code in Figure 1.2 illustrates the usage of instance variables in guards. In this code, the variable *num* records the number of items currently in the buffer. The guards specify that *Put* can execute in mutual exclusion if the buffer is not full; similarly *Get* can execute in mutual exclusion if the buffer is not empty.

Several well-known synchronisation policies schedule invocations based on the relative value of parameters. Although Guide permits parameters of an operation to be used in guards, this is of little practical use since guards can compare a parameter only to, say, an instance variable or a constant. No facility is provided to compare a parameter with the corresponding parameter of other invocations and thus Guide cannot directly implement policies that schedule invocations based on the relative values of their parameters.

```
class Buffer[elem, Size] {
  int num; ...
  Buffer() { ... }   // constructor
  Put(...) { ... }
  Get(...) { ... }
synchronisation
  Put: exec(Put, Get) = 0 and num < Size;
  Get: exec(Put, Get) = 0 and num > 0;
}
```

Figure 1.2: The Bounded Buffer

1.3.2 Dragoon

DRAGOON's synchronisation mechanism [Atk90] is similar to that of Guide in that it is based on guards that contain expressions involving synchronisation counters and instance variables. Unlike Guide, DRAGOON does not permit parameters to be used in guards. However, by far the biggest difference concerns the placement of guards. In Guide, guards are written in the same class as sequential operations, while in DRAGOON they are written in a separate class—a *behavioural class* in DRAGOON parlance.

DRAGOON separates synchronisation code from sequential code in this manner in an attempt to tackle the so-called "inheritance anomaly" (a discussion of which is deferred to the next chapter). For now, we just present an overview of how behavioural classes are used in practice.

```
behavioural class ReadersWriter {
rules ReadOps, WriteOps;
where
  ReadOps: exec(WriteOps) = 0;
  WriteOps: exec(ReadOps, WriteOps) = 0;
}
```

Figure 1.3: Dragoon-style *ReadersWriter* behavioural class

Figure 1.3 shows a behavioural class (expressed in the pseudo-code notation of this thesis rather than actual DRAGOON syntax) that expresses the basic readers/writer policy. This policy is expressed not in terms of the actual operations of a class but rather in terms of the abstract operations, *ReadOps* and *WriteOps*, which it **rules**.

The guards of a behavioural class are applied to the operations of a sequential class by a technique known as *behavioural inheritance*. Consider a class, *Foo*, that implements three sequential operations, A, B and C. If A and B are read-style operations and C is a write-style operation then the *ReadersWriter* behavioural class could be applied to *Foo* as shown

in Figure 1.4. The result is known as a *behavioured* class.

```
class SynchronisedFoo inherits Foo, ReadersWriter {
where
   A, B → ReadOps
   C → WriteOps;
}
```

Figure 1.4: Dragoon-style behavioural inheritance

Since behavioural classes are written in a manner that is independent of any particular sequential class, guards cannot directly access the instance variables of a sequential class to which they are applied. Instead, a guard invokes a boolean operation, the return value of which indicates the state of the object being synchronised. This is illustrated in Figure 1.5 which implements a bounded buffer policy. During behavioural inheritance, the programmer will map the two abstract operations, *IsEmpty* and *IsFull*, onto actual operations of the sequential class, the return values of which indicate if the buffer is empty or full, respectively. These operations will then be invoked as part of evaluating the guards for *PutOps* and *GetOps*. Note that in this behavioural class, *mutex* (mutual exclusion) is a shorthand notation meaning:

exec(all-operations-in-this-class) = 0

```
behavioural class BoundedBuffer {
rules PutOps, GetOps, IsEmpty, IsFull;
where
   PutOps: mutex and not IsFull;
   GetOps: mutex and not IsEmpty;
   IsFull: mutex;
   IsEmpty: mutex;
}
```

Figure 1.5: Dragoon-style *BoundedBuffer* behavioural class

1.3.3 Scheduling Predicates

We finish off this discussion of counter-based mechanisms with an overview of Scheduling Predicates (SP) [MWBD91]. SP was developed by the author and forms part of the contribution of this thesis. As such, it might seem strange to discuss SP in the introductory part of this thesis. However, we feel it is appropriate for two reasons. Firstly, SP is closely related to synchronisation counters (as we illustrate later in this section) and so it fits into a discussion of counter-based mechanisms. Secondly, the next chapter uses SP constructs when

introducing one of the issues that will be addressed in this thesis so it will help if the reader is familiar with SP before then.

Scheduling Predicates provide a way for guards to compare the parameters, and relative arrival time of invocations, thus making it easy to implement a range of scheduling policies. We illustrate SP by example. The following guard is used to impose a FCFS (first-come, first-served) policy on an operation, *Foo*:

Foo: *exec*(Foo) = 0 **and**
there_is_no(f **in** *waiting*(Foo): f.arr_time < *this_inv*.arr_time);

The variable f is used to iterate over the set of invocations *waiting* to execute operation *Foo*, and *this_inv* refers to the invocation for which the guard is being evaluated. *Arr_time* is an automatically maintained variable which denotes the time an invocation arrived at the object. Thus, the above guard says that when an invocation is made upon operation *Foo*, it will be blocked until the following two conditions are met:

1. *exec*(Foo) = 0 (i.e., there are no current executions of *Foo*), and
2. there is no invocation, f, currently waiting to execute *Foo*, which has a smaller *arr_time* attribute, i.e., no pending invocation has arrived at the object before this invocation.

If a formal parameter of operation *Foo* was used in place of *arr_time* in the above guard then scheduling would be based on the value of that parameter in *waiting* invocations.

Scheduling Predicates can iterate over not just invocations that are *waiting* to execute, but also those that are actually *executing*.[2] We illustrate this by solving the Dining Philosophers problem. In this problem, a philosopher may eat if there are no other philosophers eating with a fork which she herself needs. This condition can be restated as: a philosopher may eat if there are no other philosophers already eating to her left, to her right or at the table position she wants to use. This is implemented directly in Figure 1.6.

```
class Table {
  Eat(pos: 0..4) { ... }
synchronisation
  #define Right(k)       ((k + 1) mod 5)
  #define Left(k)        ((k + 4) mod 5)
  #define ShareForks(i, j)       (Right(i) = j or i = j or Left(i) = j)
  Eat: there_is_no(p executing Eat: ShareForks(p.pos, this_inv.pos));
}
```

Figure 1.6: Scheduling Predicate solution to Dining Philosophers problem

[2] An implementation of SP will have the ability to iterate over *executing* invocations only if the host language permits concurrency within objects.

1.3.3.1 Other Scheduling Predicates

The *there_is_no* predicate is akin to $\nexists$ as commonly used in predicate logic. The companion predicates, *there_exists* ($\exists$) and *for_all* ($\forall$) are also available and programmers can express a desired scheduling policy in whichever predicate feels the most natural for the task. All these predicates can be thought of as syntactic sugar for a more fundamental function, *count*, which returns an integer indicating how many invocations satisfy *count*'s condition. The following equality holds:

$$\textit{there\_is\_no}(\text{p } \textbf{in } \textit{waiting}(\text{Op}) : \textit{condition}) \equiv \textit{count}(\text{p } \textit{waiting}(\text{Op}) : \textit{condition}) = 0$$

A more in-depth discussion of these other scheduling predicates can be found elsewhere [MWBD91].

1.3.3.2 Relationship to Synchronisation Counters

SP supports the two "verbs" *waiting* and *executing*. The relationship between these "verbs" and the corresponding synchronisation counters can be expressed as follows:

$$\textit{wait}(\text{Op}) \equiv \textit{count}(\text{p } \textit{waiting}(\text{Op}) : \textbf{true})$$
$$\textit{exec}(\text{Op}) \equiv \textit{count}(\text{p } \textit{executing}(\text{Op}) : \textbf{true})$$

If the other verbs *arrived*, *started* and *terminated* were supported[3] then these would also be related to their corresponding counters as follows:

$$\textit{arrival}(\text{Op}) \equiv \textit{count}(\text{p } \textit{arrived}(\text{Op}) : \textbf{true})$$
$$\textit{start}(\text{Op}) \equiv \textit{count}(\text{p } \textit{started}(\text{Op}) : \textbf{true})$$
$$\textit{term}(\text{Op}) \equiv \textit{count}(\text{p } \textit{terminated}(\text{Op}) : \textbf{true})$$

Thus we see that both synchronisation counters and scheduling predicates can be considered to be a form of syntactic sugar for the *count* function.

1.4 Path Expressions

Path Expressions [CH73] use a notation based on regular expressions to specify synchronisation constraints on operations. They take the form:

path *reg expr* **end**

A path expression loops to repeatedly process the regular expression. Within *reg expr*, operation names denote themselves, a comma (",") denotes a choice and a semicolon (";") denotes a sequence. Also, mutual exclusion is the default in path expressions. Thus for example, the following path expression states that either operation A or B can execute (but not both at the same time):

[3] It would be infeasible for an implementation of SP to support the *arrived*, *started* and *terminated* verbs since this would require maintaining details of invocations indefinitely and the amount of information to be maintained would grow indefinitely large.

path A , B **end**

Braces ("{...}") remove the mutual exclusion constraint from whatever lies inside them. For example, the following path expression specifies that at any one time there can be either multiple executions of operation A or a single execution of operation B:

path { A } , B **end**

If A is a read-style operation and B is a write-style operation then it is clear that the above path expression implements the basic readers/writer policy.

The basic Path Expressions mechanism has spawned a number of variations including Open Path Expressions [CK80] and Predicate Path Expressions [And79].

1.5 Eiffel||

In the Eiffel|| language [Car93], objects are either *process* or *passive*.

Passive objects are local to the process object in which they were created.

Process objects have their own thread which both schedules pending invocations and services them. There is no concurrency *within* a process object. However, there is concurrency *between* objects since multiple process objects can execute concurrently with respect to each other.

A process object is an instance of a class that inherits from the standard Eiffel|| class *PROCESS*. This class provides an operation, *Live*, that is executed when an object is created. *Live* enters an infinite loop in which it chooses a pending invocation to be serviced and services it on behalf of the client. The actual policy implemented by operation *Live* in class *PROCESS* is first-come, first-served. However, subclasses can re-implement *Live* in order to use a different scheduling policy.

```
class BoundedBuffer inherits PROCESS {
  Put(...) { ... }
  Get(...) { ... }
  Live()
  { while true do
      if "the buffer is not full" then serve_oldest(Put); endif
      if "the buffer is not empty" then serve_oldest(Get); endif
      wait_for_an_invocation();
    end
  }
}
```

Figure 1.7: An Eiffel||-style bounded buffer

As well as providing *Live*, the *PROCESS* class also provides some operations to help manage the list of pending invocations. For example, the operation *serve_oldest(Foo)* will

search for any pending invocations of operation *Foo* and service the oldest one it finds. If there are no pending invocations for *Foo* then it returns straight away rather than block until a *Foo* invocation arrives.

The bounded buffer class in Figure 1.7 inherits from *PROCESS* and re-implements the *Live* operation to use a policy suitable for its needs. This shows the *serve_oldest* operation in use. The code also utilises another operation, *wait_for_an_invocation*, in order to avoid "busy-waiting" if there are no pending invocations.

1.6 Enabled-sets

Kafura and Lee [KL89] introduced the concept of *behaviour abstraction*. Tomlinson and Singh [TS89] enhanced this concept and renamed it *Enabled-sets*. The basic concept, by whatever name, has since been incorporated into numerous synchronisation mechanisms.

With Enabled-sets, a class specifies all the synchronisation *states* in which it is possible for an instance of that class to find itself. For example, at various times a bounded buffer might be in any one of the following states: *empty*, *partial* (i.e., partially full) or *full*. Each of these states enables a particular set of operations to execute. For example, when in the *partial* state, both *Put* and *Get* can execute. Similarly, the *empty* state enables only *Put* to execute (since one cannot *Get* from an empty buffer) and the *full* state enables only *Get* to execute (since one cannot *Put* any more items into an already full buffer). Thus the complete set of states for a bounded buffer, and the operations enabled in each of these states, might be expressed as follows:

state empty **enables** {Put}
state full **enables** {Get}
state partial **enables** {Put, Get}

Alternatively, the set of operations for the *partial* state could be expressed in terms of the sets associated with the *empty* and *full* states, using the following notation:

state partial **enables** empty.ops + full.ops

Within the body of an operation, a **become** statement is used to specify which state the object should enter next (and hence which operations are permitted to execute). For example, the statement "**become** full" indicates that a bounded buffer object is in the *full* state and that only the *Get* operation can execute. An example of a bounded buffer class implemented with Enabled-sets is shown in Figure 1.8.

The main purpose of Enabled-sets is that if a subclass introduces a new operation then this operation could be added to (some of) the sets associated with the synchronisation states inherited from the parent class. In this way, it was hoped, Enabled-sets would permit the synchronisation code of a base class to adapt to the needs of subclasses and hence tackle the "inheritance anomaly" (which will be discussed in the next chapter).

```
class BoundedBuffer inherits PROCESS {
  state empty enables {Put};
  state full enables {Get};
  state partial enables {Put, Get};
  BoundedBuffer(...)      // constructor
  { ...
    become empty;
  }
  Put(...)
  { "add item to end of buffer";
    if "buffer is full" then become full; else become partial; endif
  }
  Get(...)
  { result := "remove item from start of buffer";
    if "buffer is empty" then become empty; else become partial; endif
    return result;
  }
}
```

Figure 1.8: A bounded buffer written with the aid of Enabled-sets

1.7 Structure of this Thesis

This thesis is divided into five parts. Part I (which you are currently reading) and Part V are the introduction and conclusions, respectively. The middle three parts contain the contributions of this thesis.

- Part II presents a paradigm for developing powerful synchronisation mechanisms.
- Part III shows how it is possible to provide comprehensive language support for generic synchronisation policies.
- Part IV analyses the problems regarding the use of inheritance in COOPLs.

Details of the structure of each part of the thesis can be found at the start of the relevant parts.

Chapter 2

Aims of this Thesis

This thesis makes contributions in several areas: (i) expressive power of synchronisation mechanisms; (ii) the problem of synchronisation mechanisms having unsafe access to instance variables; (iii) language support for generic synchronisation policies; and (iv) problems with the use of inheritance in COOPLs.

For each of these areas, which are discussed in Sections 2.1 through to Section 2.4, we discuss the key issues of the area and then state the contributions that this thesis makes in addressing these issues.

2.1 Expressive Power

Hoare [Hoa74] shows that Semaphores and Monitors can be implemented in terms of one another. A great many other synchronisation mechanisms can also implement Semaphores, and can in turn be implemented by Semaphores. This means that, at a *theoretical* level, all these synchronisation mechanisms have equal power since each is equivalent in power to Semaphores.

However, at a *practical* level, one synchronisation mechanism might be able to implement a particular synchronisation policy (or range of policies) more easily than another synchronisation mechanism. The term "expressive power" refers to the ability of a synchronisation mechanism to *easily* implement a range of synchronisation policies: the wider the range of synchronisation policies a mechanism can *easily* implement, the greater its expressive power is said to be.

It is common for the designers of a synchronisation mechanism to illustrate its expressive power by implementing several well-known synchronisation policies. Unfortunately, this does not provide an objective criteria for gauging expressive power since the synchronisation policies commonly used are usually chosen because they happen to be well-known rather than because of any principles they might embody.

Bloom [Blo79] proposes an alternative to this. She lists six types of information that a synchronisation mechanism should have access to in order for it to have good expressive

power. The six types of information are as follows:

1. The name of the invoked operation.
2. The relative arrival time of invocations.
3. Invocation parameters.
4. The "synchronisation state" of the resource, i.e., information about how many processes are *currently* accessing the object.
5. The local state of the object, i.e., instance variables.
6. History information; this is similar to the "synchronisation state," except that it refers to operation invocations that have already terminated. Bloom notes that since past invocations will likely have affected instance variables, the "history information" category is often interchangeable with the "instance variables" category.

It is relatively easy to determine the types of information to which a particular synchronisation mechanism has access. For example, a synchronisation mechanism that permits synchronisation counters to be used in guards would have access to:

1. The name of the invoked operation—each operation has its own guard and set of synchronisation counters associated with it.
4. The "synchronisation state" of the resource—the *exec* and *wait* counters provide this information.
6. Some history information—this is provided by the *term* counter.

If this analysis is carried out for several synchronisation mechanisms then it allows these mechanisms to be compared with each other.

For example, a synchronisation mechanism, x, might have access to information types 1, 2, 4 and 6, while another synchronisation mechanism, y, might have access to information types 1, 2 and 4. In this case, it is clear that mechanism x has more expressive power than y. If instead y had access to information types 1, 3, 5 and 6 then x and y would have *different* kinds of expressive power, with neither one being clearly superior to the other. Obviously, it is desirable for a synchronisation mechanism to have a high degree of expressive power. However, expressive power often comes at a price, as we will discuss in Sections 2.1.1 to 2.1.4. Then in Section 2.1.5 we will state the contributions that this thesis makes in the area of expressive power.

Note that Bloom's list can be used not only to evaluate the expressive power of a synchronisation mechanism but also to evaluate the type of expressive power required to implement a particular synchronisation policy. For example, to implement a bounded buffer requires access to the following types of information:

1. The name of the invoked operation—since *Put* and *Get* have different synchronisation constraints.

4. The "synchronisation state of the object"—since some form of mutual exclusion will be required.

5/6. Whether the buffer is *empty*, *full* or in between could be recorded by either an instance variable (information type 5) or a synchronisation variable recording this "history information" (information type 6).

One could apply a similar analysis to a number of synchronisation policies and then choose a subset of these such that, between them, they exercise each of Bloom's six types of information. This could be used as a more objective test suite for evaluating the expressive power of synchronisation mechanisms than an ad hoc collection of well-known synchronisation policies.

2.1.1 Creeping Featurism and Complexity

As already shown, a guard-based mechanism employing synchronisation counters has access to three out of Bloom's six types of information.

In order to increase the expressive power of such a mechanism, a language designer might add other constructs. Perhaps the guards will be allowed to access instance variables along with synchronisation counters. Also, a **fcfs** keyword might be added to specify that pending invocations are to be serviced by their relative arrival times. Finally, another construct might be added that permits scheduling based on the value of a parameter.

The resulting synchronisation mechanism has access to all six of Bloom's types of information so it obviously has greater expressive power than the original mechanism. However, it is also a great deal more complex since there are now *four* constructs that can be used in guards in contrast to the original mechanism's one.

Aside from burdening programmers with new features to learn, these new constructs may introduce problems to the language. For instance:

- If a guard is reading an instance variable while that variable is being updated by an operation then the guard might see it in an inconsistent state. (See Section 2.2 for an extensive discussion about this problem.)
- It may not be possible to use certain combinations of constructs. For instance, it may not be possible to combine scheduling based on parameters with the **fcfs** construct, thus hindering the implementation of a policy such as, say, a Shortest Job Next Scheduler with FCFS sub-ordering.

Also, with so many constructs, each added to cope with a specific problem, some constructs might be *too* specific and lack flexibility, thus not providing as much expressive power as they might have done. For example, the construct for scheduling might be able to schedule invocations for *one*, but not *several*, operations. Also, since scheduling based on parameters is similar to scheduling based on relative arrival times perhaps a single construct would have served both purposes (as it does in SP).

Thus, unless the new constructs are carefully chosen so that they are as general-purpose as possible and integrate well both with each other and with existing language constructs, the language may end up being overly complex, idiosyncratic and lacking in expressive power *despite* having access to all six of Bloom's types of information.

An example of this ineffective creeping featurism is provided by the evolution of the synchronisation mechanism in the Guide language. As discussed in the previous chapter, an early language definition [DDR+91] permitted guards to access: (i) synchronisation counters, (ii) parameters, and (iii) instance variables. More recent papers [RR92, Riv92] propose four extensions to Guide's guard-based mechanism, *plus* a completely different synchronisation mechanism (similar to Path Expressions) to be used in *types*, the expressive power of which is less than that already provided by guards in *classes* (which implement types).

Another example is provided by DRAGOON. The synchronisation mechanism described in Atkinson's thesis [Atk90] permits guards to access just synchronisation counters and instance variables. A more recent paper [ACR92] outlines *five* different proposed extensions to the basic mechanism.

2.1.2 Graceful Degradation of Expressive Power

We do not know of any synchronisation mechanism that has *infinite* expressive power, i.e., that can implement all possible synchronisation policies easily. So, since every synchronisation mechanism has *finite* expressive power, it stands to reason that for any given mechanism there may be a policy, $\mathcal{P}$, that is *just* within its expressive power and a similar, though slightly more complex policy, $\mathcal{P}'$, that is beyond its expressive power. This prompts the question: will the difficulty of implementing $\mathcal{P}'$ be *proportional* to how far it is beyond the expressive power of the synchronisation mechanism? Or will there be a sudden jump in difficulty? If the former is the case then this suggests that the implementation of $\mathcal{P}'$ will bear some similarity to the implementation of $\mathcal{P}$, while the latter suggests that there will be little, if any, similarity between the two implementations.

A classic example of this sudden jump in the difficulty level of implementing related synchronisation policies is provided by the Path Expressions implementation of various readers/writer policies [CH73, pg. 93–94]. While the basic readers/writer policy can be expressed easily via a single path expression, other variations require multiple path expressions and the introduction of what Bloom [Blo79, pg. 28] calls "synchronisation procedures."

Another example is provided in the Guide implementation of various readers/writer policies [DDR+91]. While guards are able to elegantly express several variations—basic readers/writer, readers priority and writers priority—the implementation of a FCFS variant requires that guards be combined with, what are in effect, synchronisation procedures, and the resulting code bears little resemblance to the code used for, say, the basic readers/writer policy.

As a final example, consider Scheduling Predicates which can implement the Shortest Job Next scheduler easily [MWBD91]. This policy is *unfair* since it is possible for a long job to be

skipped over indefinitely by a continuous stream of shorter jobs. One way to remedy this is to decrement the "length" of a job whenever it is skipped over by a shorter job, thus decreasing the likelihood of it being skipped over again. However, Scheduling Predicates does not have the expressive power to directly implement this starvation-free version of the Shortest Job Next scheduler and programmers would have to resort to explicitly maintaining their own queues of pending jobs.

2.1.3 Declarative Vs. Procedural Mechanisms

Many synchronisation mechanisms (including Path Expressions and guard-based mechanisms) are *declarative* in nature. By this we mean that programmers can use the mechanism simply to specify the synchronisation policy desired. They need not worry about implementation details because, ideally, the implementation will be derived automatically from the specification.

However, while declarative mechanisms can often give elegant, concise solutions to some synchronisation problems, they generally have limited expressive power because they cannot express algorithms. Furthermore, this inability to express algorithms usually means that declarative mechanisms do not "degrade gracefully" when faced with a synchronisation policy that is beyond their expressive power. Rather, they generally need to utilise synchronisation procedures to implement policies beyond their expressive power; this usually results in an abrupt increase in difficulty of use. An intuitive, graphical representation of this is shown in Figure 2.1.

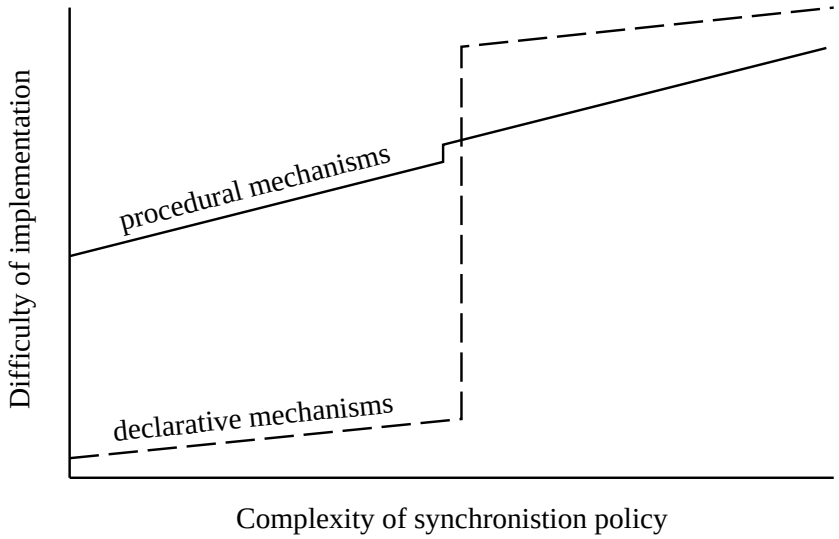

Figure 2.1: Graphical comparison of declarative and procedural synchronisation mechanisms

In contrast to declarative mechanisms, some mechanisms are *procedural*. These combine synchronisation primitives with sequential flow control constructs and data structures, al-

lowing programmers to implement synchronisation policies through algorithms. Examples of procedural mechanisms include Eiffel|| (introduced in Section 1.5 on page 11), and Monitors [Hoa74]. The synchronisation primitives provided are usually at a low-level which often results in verbose and intricate implementations of synchronisation policies. So, from this point of view one might argue that they do not have good expressive power. However, as Figure 2.1 shows, in being able to utilise algorithms, procedural mechanisms generally degrade more gracefully than declarative mechanisms and often complex scheduling policies are not much more difficult to implement than simpler ones.

2.1.4 Modularity

The sequential code and synchronisation code of a class serve different purposes: the former implements the services (operations) of the class while the latter ensures data consistency in the face of concurrent access (and possibly also imposes a scheduling order upon pending invocations). Because they serve different purposes, the synchronisation code and sequential code of a class should be separated from one another [Blo79, pg. 25].

Some synchronisation mechanisms keep synchronisation code separated from sequential code, by default, and thus obtain modularity. However, if a synchronisation mechanism has limited expressive power then programmers may resort to mixing synchronisation code with sequential code in an effort to implement policies beyond the mechanism's native expressive power. An example of this mixing of synchronisation code with sequential code is the "synchronisation procedures" of Path Expressions, which we briefly mentioned in Section 2.1.2.

While some synchronisation mechanisms provide modularity by default and only violate it in order to implement complex synchronisation policies, some other mechanisms routinely mix synchronisation code with sequential code. An example of this is Enabled-sets (introduced in Section 1.6 on page 12) in which the transition from the current synchronisation state to the next is achieved by code embedded inside the sequential code of operations.

2.1.5 Contributions

Part II of this thesis defines a new paradigm, Sos, for synchronisation mechanisms and it is this paradigm that makes several contributions in the area of expressive power. The contributions it makes are as follows.

The Sos paradigm shows that all of Bloom's six types of information are derived from a single source of information. Thus, a synchronisation mechanism that has access to this primary source of information can have similar expressive power to another synchronisation mechanism that has access to Bloom's six derivative types of information. A benefit of this finding is that fewer constructs are likely to be needed in order to access the single, primary source of information than the multiple derivative sources of information; hence, a synchronisation mechanism can have good expressive power without undue complexity or creeping featurism.

The SOS paradigm also fully separates synchronisation code from sequential code, thus ensuring that modularity does not have to be sacrificed for good expressive power.

Part II of this thesis also introduces a sample synchronisation mechanism, ESP, to illustrate the concepts of SOS. Through ESP, we verify the claimed benefits of the SOS paradigm, i.e., it offers good expressive power without sacrificing modularity. An additional benefit that ESP offers is that it merges declarative and procedural programming styles and, in so doing, degrades gracefully when used to implement more complex synchronisation policies.

2.2 Unsafe Access to Instance Variables

Consider an object that can be accessed concurrently by several processes. Some mechanism must be used to protect the object in the face of concurrent access—otherwise its data might become inconsistent. Encapsulation requires that the code to protect an object should be placed *within* the object itself rather than be spread out among its clients [Blo79]. We refer to this code as *synchronisation code*.

It is believed that access to the instance variables of an object by its synchronisation code is needed in order to implement many synchronisation policies [Blo79]. This introduces an obvious difficulty. The synchronisation code must not read an instance variable while that variable is being updated by an operation, otherwise the synchronisation code might see the variable in an inconsistent state.

Surprisingly, many languages and systems (e.g., Guide [DDR+91], PLOOC [Tho92], Dooji [Tho94], CEiffel [Löh93], VCP [GC92], Conditional Path Expressions [GW91], SR [And81], Dale [Cou94] Demeter [LL94] Composition Filters [Ber94] and DRAGOON [Atk90]) allow synchronisation code to access instance variables but do *not* provide any means to ensure that this access is performed in a safe manner.

In Sections 2.2.1 and 2.2.2 we review some existing approaches to allowing instance variables to be accessed safely by synchronisation code. (In these sections, synchronisation code is represented in the form of guards; however, the principles discussed apply to non guard-based mechanisms too.) Then in Section 2.2.3 we state the contribution that this thesis makes in the area of instance variables being accessed unsafely by synchronisation code.

2.2.1 Reduction of Concurrency

Consider an instance variable, $\mathcal{V}$, which is read by some guards. Let $\mathcal{G}$ be the set of guards which read $\mathcal{V}$, and $\mathcal{O}$ be the set of operations which update $\mathcal{V}$. One way to guarantee that the guards in the set $\mathcal{G}$ will always evaluate to a consistent value, even while $\mathcal{V}$ is being updated, is to ensure that the condition "$exec(\mathcal{O}) = 0$" is a conjunctive of each guard in $\mathcal{G}$.

Of course, since operations in the set $\mathcal{O}$ all update $\mathcal{V}$, $\mathcal{O}$'s guards will normally contain the conjunctive "$exec(\mathcal{O}) = 0$" (to prevent multiple writers concurrently updating $\mathcal{V}$). If this is so and if $\mathcal{G}$ is a (not necessarily strict) subset of $\mathcal{O}$ then it follows that the guards in set

$\mathcal{G}$ will *already* contain the required conjunctive. In this case, the problem is solved without any intervention by the programmer.

Consider the bounded buffer code in Figure 2.2. For $\mathcal{V} = num$, we note the following:

$\mathcal{G} = \{Put, Get\}$
$\mathcal{O} = \{Put, Get\}$

Since $\mathcal{G} = O$ (and thus $\mathcal{G}$ is a subset of $\mathcal{O}$), the problem is solved naturally.

```
class Buffer[elem, Size] {
  int   get_index, num; elem data[Size];
  Buffer() { get_index := 0;   num := 0; } // initialisation
  Put(...) { ... "update num"; ... }
  Get(...) { ... "update get_index & num"; ... }
synchronisation
  Put: exec(Put, Get) = 0 and num < Size;
  Get: exec(Put, Get) = 0 and num > 0;
}
```

Figure 2.2: The Bounded Buffer

However, one cannot rely on the problem always being solved naturally, as the following illustrates.

Dynamic Priority Print Queue

This problem is an example where $\mathcal{G}$ is not a subset of $\mathcal{O}$, and extra conjunctives must be added to guards to ensure their consistent evaluation. The problem description is as follows:

> At a college, a printer is accessed by undergraduate students from first to fourth year, graduate students and members of staff. The printer queue is priority based (each group has its own priority) and there is FCFS ordering within a priority level. From time to time, the system manager may change the group priorities.

An attempt at implementing this is given in Figure 2.3. The error in this code becomes apparent if we consider the guard of a *Print* invocation evaluating while *UpdatePriority* is executing: because the variable *priority[]* is being updated it is potentially in an inconsistent state which could result in the guard being incorrectly evaluated.

For $\mathcal{V} = priority[]$, we note the following:

$\mathcal{G} = \{Print\}$
$\mathcal{O} = \{UpdatePriority\}$

As discussed earlier, to fix this we must make "$exec(\mathcal{O}) = 0$" a conjunctive of $\mathcal{G}$. The resultant guard for $\mathcal{G}$ (i.e., *Print*) is then:

Print: *exec*(Print, UpdatePriority) = 0 **and** *there_is_no*(...)

```
type GroupId = (one, two, three, four, grad, staff);

class Printer {
  int   priority[GroupId];
  Printer() { "initialise priority[] to appropriate values"; }
  UpdatePriority(GroupId gid, int NewPriority) { priority[gid] := NewPriority; }
  Print(GroupId gid, string FileName) { ... }
synchronisation
  UpdatePriority: exec(UpdatePriority) = 0;
  Print: exec(Print) = 0 and there_is_no(p in waiting(Print):
            priority[p.gid] > priority[this_inv.gid] or
            priority[p.gid] = priority[this_inv.gid] and p.arr_time < this_inv.arr_time);
}
```

Figure 2.3: An attempt at the Dynamic Priority Print Queue problem

2.2.1.1 Discussion

This approach has the advantage that it does not require any changes to existing languages or synchronisation mechanisms. However, it has two disadvantages:

1. It solves the problem by reducing the potential for concurrency within an object.
2. Adding conjunctives to guards to ensure that they may safely access instance variables is error prone and tedious, and programmers may forget to check the guards whenever a modification is made to the source code of the class.

2.2.2 The Guide Proposal

For some time, Guide [DDR+91] allowed instance variables to appear in guards and ignored the problems which this raised. However, a more recent paper [Riv92] proposes a way to allow instance variables to be used safely in synchronisation guards (a similar scheme has also been proposed for Dale [Cou94, pg. 193]). This is achieved in the following manner:

1. A lock is used to ensure that the evaluation of synchronisation guards and the updating of synchronisation counters takes place atomically. Let us call this lock "lock_{sync}".
2. If a variable, $\mathcal{V}$, is used in a guard then an assignment to $\mathcal{V}$ is replaced with the following code:

```
tmp := right-hand-side of the assignment;
acquire lock_sync;
  V := tmp;
  re-evaluate guards which rely on V;
release lock_sync;
```

We shall use some examples to show that this technique is inadequate. For consistency, the examples will be presented in the syntax used throughout this thesis rather than in the syntax of Guide.

Dynamic Priority Print Queue

Consider the attempt at the dynamic priority printer shown previously in Figure 2.3. When we originally discussed this in Section 2.2.1, we showed the error caused by the possibility of *priority[]* being updated while the guard for *Print* was being evaluated. Under the semantics of the Guide proposal, this error disappears without the programmer having to alter the guards.

However, consider what would happen if the *UpdatePriority* operation was rewritten so that it modified the priority of *all* the groups rather than just a single one, as shown below:

```
UpdatePriority(int NewPriority[GroupId])
var GroupId   index;
{
  for index in one..staff do
    priority[index] := NewPriority[index];
  end
}
```

If the code generated by the compiler acquires and releases lock_{sync} *inside* the body of the **for** loop then the bug reappears since the *priority[]* array, as a whole, will not be consistent while guards are being evaluated. To ensure consistency of the array, the compiler would have to generate code which would place the acquisition and release of lock_{sync} around the entire **for** loop.

The compiler writer might be able to take into account this interaction of **for** loops and the usage of lock_{sync}, but, unfortunately, the problem is more general than that. The compiler would have to be able to generate code so that the acquisition and release of lock_{sync} would surround *any* arbitrary sequence of statements which updated an equally arbitrary *set* of variables, since the synchronisation code might rely on the set of variables *as a whole* being consistent.

The dynamic priority print queue is not an isolated example for which the Guide proposal proves insufficient, as the next example shows.

Dining Philosophers

This well-known problem concerns a table with five seats and five chopsticks—one at each seat position. Because two chopsticks are required for eating, philosophers use the chopstick at their own seat and also the chopstick at the seat to their right. Neighbouring philosophers cannot simultaneously share the chopstick which is common to them. In simulating the action

```
type TablePositon is int subrange(0..4);
class DiningPhilTable {
   boolean    chopstick_avail[TablePosition];
   DiningPhilTable()       // constructor
   { "set all chopstick_avail[] to true"; }
   Eat(TablePosition pos)
  {  chopstick_avail[pos] := false;
     chopstick_avail[(pos+1) mod 5] := false;
     ...   // "real" Eat code here
     chopstick_avail[pos] := true;
     chopstick_avail[(pos+1) mod 5] := true;
   }
synchronisation
   Eat: chopstick_avail[this_inv.pos] and chopstick_avail[(this_inv.pos+1) mod 5];
}
```

Figure 2.4: First attempt at Guide solution to the Dining Philosophers problem

at the table, one must ensure against deadlock as might happen if, say, all five philosophers picked up their own chopsticks simultaneously, only to discover that the chopstick to their right was already held by their neighbour.

Figure 2.4 shows a first attempt at solving this. The guard on *Eat* reflects the problem description and is quite intuitive. However, this solution is flawed because, when an invocation starts executing *Eat*, there is no guarantee that it will update the status of its two chopsticks before another invocation arrives and evaluates its own guard. If this latter invocation is for a neighbouring seat of the first invocation then the two invocations will attempt to simultaneously share chopsticks—which is forbidden by the problem description.

To fix this flaw, we factorize any code which updates *chopstick_avail[]* out of *Eat* and into the new operations *PickUp* and *PutDown*. The resultant code is shown in Figure 2.5. However, this solution works, not because it makes use of Guide's proposed semantics, but rather because it falls back to using the technique discussed in Section 2.2.1. You can see this by noting that, for $\mathcal{V} = \textit{chopstick\_avail[]}$, we have:

$$\mathcal{G} = \{\textit{PickUp}, \textit{PutDown}\}$$
$$\mathcal{O} = \{\textit{PickUp}, \textit{PutDown}\}$$

2.2.2.1 Discussion

From these examples we can see that Guide's new proposal offers only a partial solution to the problem. If a single instance variable is used in the guards then Guide's proposal may help. However, it fails if more than one instance variable (or if a compound variable, such as an array or record) is used in the synchronisation code. In this case, the programmer must

```
type TablePositon is int subrange(0..4);
class DiningPhilTable {
   boolean    chopstick_avail[TablePosition];
   DiningPhilTable()       // constructor
   { "set all chopstick_avail[] to true"; }
   Eat(TablePosition pos)
   {  PickUp(pos, (pos + 1) mod 5);
      ...   // real "eat" code here
      PutDown(pos, (pos + 1) mod 5);
   }
   PickUp(TablePosition i, TablePosition j)
   {  chopstick_avail[i] := false;
      chopstick_avail[j] := false;
   }
   PutDown(TablePosition i, TablePosition j)
   {  chopstick_avail[i] := true;
      chopstick_avail[j] := true;
   }
synchronisation
   PickUp: exec(PickUp, PutDown) = 0 and chopstick_avail[this_inv.i]
            and chopstick_avail[this_inv.j];
   PutDown: exec(PickUp, PutDown) = 0;
   // no guard needed for "Eat"
}
```

Figure 2.5: Guide solution to the Dining Philosophers problem

revert to the technique discussed in Section 2.2.1, i.e., reduce the potential for concurrency within objects.

2.2.3 Contribution

As we have said before, it is commonly believed that synchronisation code needs to access the instance variables of an object in order to implement many synchronisation policies. Through the SOS paradigm, introduced in Part II of this thesis, we will show that this belief is incorrect. In particular, we will show that a synchronisation mechanism can maintain its own variables and in doing so can implement synchronisation policies without having to access instance variables.

2.3 Generic Synchronisation Policies

Language support for generic data types is becoming more and more common, especially in object-oriented languages. Thus, you might see data types such as:

type List[T:Any[1]]
type Set[T:Comparable]

In general, generic data types take one or more formal parameters that serve as place-holders when instantiating the generic type upon an actual type. For instance, one might declare a variable of type "list of employees" as follows:

var a: List[Employee]

Generic data types promote code reuse since a particular data type need be written only once and then can be instantiated (reused) many times.

Some synchronisation policies such as mutual exclusion and readers/writer are quite common and are used in a number of different classes. This suggests the possibility of providing language support for *generic synchronisation policies* (GSPs) so that commonly used policies need be written only once.

For example, a mutual exclusion synchronisation policy might be expressed as:

policy Mutex[Ops: Set[Operation]]

In this example the formal parameter, *Ops*, represents a set of operations. Thus, given a set of operations, "{A, B, C}", the policy might be instantiated as follows:

Mutex[{A, B, C}]

Notice that this policy is instantiated upon the set value "{A, B, C}". This is in contrast to the examples of generic data types given earlier which were instantiated upon types.

As another example, a readers/writer policy might be expressed as:

policy ReadersWriter[ReadOps: Set[Operation], WriteOps: Set[Operation]]

To express some other synchronisation policies one might extend the range of parameters upon which it is possible to instantiate a policy. For example, a bounded buffer synchronisation policy might be expressed in terms of a set of put-style operations, a set of get-style operations and the *Size* of the buffer (an integer):

policy BBuf[PutOps: Set[Operation], GetOps: Set[Operation], Size: Int]

Similarly, one might extend the notation in order to be able to instantiate a synchronisation policy on parameters of operations. For example, a Shortest Job Next scheduler might be denoted by:

[1] In some object-oriented languages, *Any* is the base class from which all other classes are derived, and a class called, say, *Comparable* provides an operation that can be used to compare two objects for equality.

policy SJN[Ops: Set[Operation], Length: Parameter]

The possibility of providing language support for generic synchronisation policies raises a number of issues, including:

- How many different types of formal parameter are needed to support generic synchronisation policies? The examples given so far suggest a minimum requirement of being able to instantiate generic synchronisation policies upon sets of operations, parameters and integer constants.
- Generic synchronisation policies will be written outside the context of any particular sequential class. However, many existing synchronisation mechanisms permit synchronisation code to be mixed with sequential code in an effort to gain more expressive power. This raises the question of whether an enforced separation of synchronisation code from sequential code will limit the range of generic synchronisation policies that can be implemented.
- Is it possible to have inheritance hierarchies for generic synchronisation policies? For example, perhaps *ReadersWriter* could be a base policy and from which other policies, e.g., *ReadersPriority*, inherit.

Some existing languages attempt to provide support for generic synchronisation policies. For example, the behavioural classes of DRAGOON (introduced in Section 1.3.2 on page 7) are, in effect, generic synchronisation policies. However, the support provided in DRAGOON has some limitations, including:

- DRAGOON uses a form of inheritance to instantiate a behavioural class, and thus confuses genericity with inheritance. As we will discuss later in Section 10.1.1, this confusion can lead to name-space pollution of classes and inconsistencies in the typing system.
- Behavioural classes have access to the instance variables of a class, albeit indirectly via functions. Hence, generic synchronisation policies in DRAGOON cannot be written in a manner that is completely independent of sequential code.
- Due to constrained expressive power, behavioural classes can implement only a limited range of synchronisation policies.

Some other existing languages, e.g., Demeter [LL94], Hector [BFS93] and Parallel Objects [CL91], also provide support for generic synchronisation policies. However, the support in these languages is also of a limited nature.

While it may seem difficult to provide language support for generic synchronisation policies, there are several important benefits to be gained from providing such support. We already mentioned one benefit: code reuse.

Another benefit is that with proper language support it should be trivially easy to instantiate a policy. This introduces the possibility of having skilled programmers write libraries of generic synchronisation policies that can be used by less skilled programmers.

A related benefit is that the instantiation of a generic synchronisation policy is extremely declarative, irrespective of how the policy is actually implemented. Thus, the utility of a low-level, procedural synchronisation mechanism could be enhanced by encapsulating difficult to understand synchronisation code as generic policies which are declarative in their instantiation.

2.3.1 Contributions

Part III of this thesis is devoted to the area of generic synchronisation policies. It makes several contributions.

The issues surrounding language support for generic synchronisation policies are examined and we show that it is possible to provide comprehensive support in a straight-forward manner that does not limit the range of synchronisation policies that can be expressed.

We also show that GSPs facilitate the optimisation of synchronisation code.

2.4 Analysis of the Problems with Inheritance in COOPLs

It is well-known that there are problems with the use of inheritance in COOPLs that can hinder code reuse [TS89] [KL89] [Mat93] [Ber94]. These problems are commonly referred to in the literature as "inheritance anomalies."

This section gives an intuitive explanation of a subset of the problems (a more detailed analysis can be found in Part IV of this thesis). Then in Section 2.4.1 we state the contributions that this thesis makes in this area. Note that this section uses generic synchronisation policies as a means to to discuss a particular policy, e.g, *ReadersWriter*, independently of how it might be implemented or even what synchronisation mechanism might be used to implement it. In this way, we can discuss the problems with inheritance in a way that is independent of any particular synchronisation mechanism.

Consider a base class that contain three operations, A, B and C. If A and B examine instance variables and C updates them then a suitable synchronisation policy for this class would be:

ReadersWriter[{A, B}, {C}]

If a subclass re-implemented operation B and in doing so turned it into a write-style operation[2] then the subclass would have to change the synchronisation policy in order to accommodate this change. For example, the synchronisation code might be changed to:

[2] Some readers may think that for a subclass to re-implement an inherited read-style operation as a write-style operation would mean that the subclass is not a sub*type* of its parent. This issue is irrelevant to the present discussion since we are concerned with inheritance as a means of code reuse rather than as a subtyping mechanism. However, we note that it *is* possible for a subclass to re-implement an inherited read-style operation as a write-style operation and still be regarded as a subtype of its parent. For example, the instance variables updated by the re-implemented operation need not be inherited ones but rather instance variables declared new to the subclass.

ReadersWriter[{A}, {B, C}]

As another example, consider what might happen if, instead of re-implementing operation B, the subclass introduced a new write-style operation, D. Once again, the synchronisation code would have to be changed in order to accommodate this change in the sequential code. The new synchronisation code might be:

ReadersWriter[{A, B}, {C, D}]

Both of these examples illustrate aspects of the problems with inheritance in COOPLs. In effect, if a subclass changes its sequential code then this may invalidate, and necessitate changes to, its inherited synchronisation code. The converse can also happen. If a subclass makes changes to the inherited synchronisation code, e.g., in order to permit more internal concurrency, then this might necessitate changes to the inherited sequential code. Thus, while inheritance can be a good tool for reusing code in a sequential language, it does not always work so well in concurrent languages.

The above explanation of the problems with inheritance in COOPLs is somewhat simplified but it suffices as an introductory explanation. The ramifications of these problems can be severe, to the point of preventing programmers from being able to reuse *any* inherited code.

To understand why this is so, consider that not all languages keep synchronisation code and sequential code separate from one another. In some languages, synchronisation code is embedded inside the bodies of sequential operations. In such languages, sometimes the only way to change the synchronisation code of a class might be to re-implement one or more operations. This, of course, will result in the sequential code of the affected operations having to be re-implemented along with the synchronisation code. This might create a feedback loop: a change to the sequential code of a class necessitates a change in the inherited synchronisation code of the class which in turn necessitates a re-implementation of some of the inherited sequential operations. In some cases, this feedback loop may make it impossible to reuse *any* inherited code.

2.4.1 Contributions

In Part IV of this thesis, we analyse the problems associated with using inheritance in COOPLs and make the following contributions.

A common perception is that the problems are rooted in a conflict between synchronisation and inheritance, which suggests that the problems might be solved by designing new synchronisation mechanisms that do *not* conflict with inheritance. We show that this perception is incorrect and that the problems are, in fact, intrinsic to inheritance. This has two ramifications.

Firstly, the problems are more serious than previously thought since they are not confined to the specialised area of synchronisation but are at the heart of object-orientation. Hence,

they may show up in other areas. For example, similar problems have been noted in the area of real-time constraints [ABvdSB94].

Secondly, since the problems are intrinsic to inheritance rather than synchronisation, it follows that approaches to tackling the problems need to be focussed on, say, designing new inheritance models or alternative ways to reuse code, rather than on designing new synchronisation mechanisms.

Towards the aim of considering alternative reuse mechanisms, we show that the use of generic synchronisation policies can drastically reduce, though not entirely eliminate, the harmful effects of the problems with inheritance.

Part II

The Sos Paradigm: Power through Simplicity

Introduction to Part II

Part II presents SOS: a paradigm for the design of synchronisation mechanisms.

The concepts of the paradigm itself are presented in Chapter 3.

Then Chapter 4 introduces a sample synchronisation mechanism, ESP, which embodies the concepts of the SOS paradigm. The examples in this chapter illustrate that ESP (and hence the SOS paradigm) offers excellent expressive power while avoiding unsafe access to instance variables.

Having introduced the concepts of the ESP synchronisation mechanism, Chapter 5 then discusses the issues that arise when adding ESP to a host language. We show that many of the concepts of ESP can be expressed in existing language constructs and that a SOS-based synchronisation mechanism can be added to an existing language with a minimum of difficulty.

Finally, Chapter 6 summarises the contributions that the SOS paradigm makes and compares our work to that of other researchers.

Chapter 3

The Service-object Synchronisation (Sos) Paradigm

One of the main contributions of this thesis is a new paradigm for designing synchronisation mechanisms for object-oriented languages. The paradigm consists of four concepts, each of which is relatively ordinary and can be found in existing synchronisation mechanisms. What is unusual about the paradigm is that it supports these concepts to a much stronger degree than most existing synchronisation mechanisms. By providing extensive support for these four concepts, the paradigm offers several important benefits.

This chapter introduces the paradigm and its core concepts. The benefits that the paradigm offers will become apparent when the paradigm is put into practice—which is what most of the rest of this thesis does. As such, readers will not have an appreciation of the paradigm by the end of this chapter. That will follow later.

The paradigm presented here is called the *Service-object Synchronisation* (Sos) paradigm. It consists of the following concepts:

1. Events (and code that is executed at them). (This is discussed in Section 3.1)
2. A way to cause a pending invocation to start executing. (This is discussed in Section 3.2)
3. Access to information about Invocations. (This is discussed in Section 3.3)
4. A strict separation between (i) the code and data used for the synchronisation of an object, and (ii) the code and data of the object itself. (This is discussed in Section 3.4)

3.1 Events

The Sos paradigm provides synchronisation at the granularity of operation invocations, as opposed to, say, the finer granularity of individual statements within operations or at the coarser granularity of (collections of) objects. Perhaps a good way to introduce the Sos paradigm is to start by modelling the lifespan of a typical operation invocation. The model used is event-based.

The diagram in Figure 3.1 shows the sequence of events that take place when one object (the *client* object) invokes an operation upon another object (the *service* object).

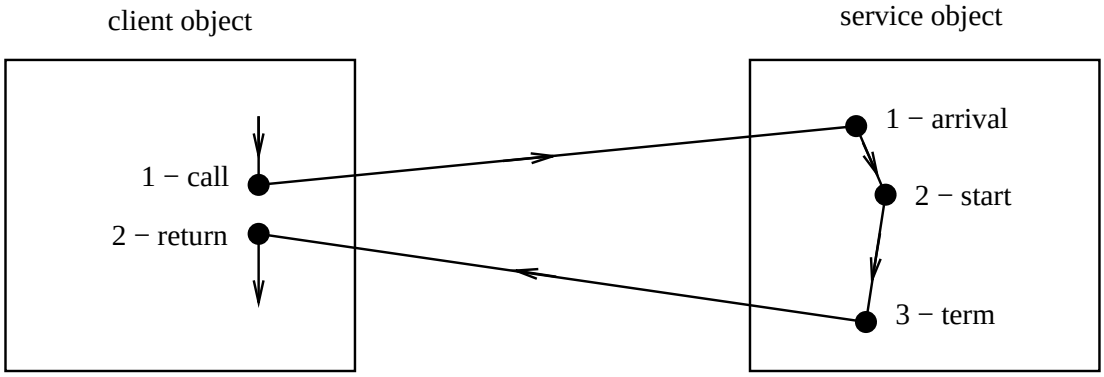

Figure 3.1: The events in the lifespan of a typical operation invocation

From the client object's perspective, there are potentially two events of interest: *call* and *return*. If the host language supports asynchronous calls then the client will be able to treat these two events independently of one another. A common form of support for this is *futures* [Hal85]. If the host language does not support asynchronous calls then the *call* and *return* events will be syntactically merged and indistinguishable from one another.

From the service object's perspective, there are potentially three events of interest: *arrival*, *start* and *term* (short for *termination*). When an invocation *arrives* it may be delayed due to synchronisation constraints. Sometime later it will *start* execution; and sometime later still it will *terminate* execution. (In a sequential system the *arrival* and *start* events would be merged.)

Note that in a distributed system the client and service objects may reside on different nodes and hence there may be a time-lag between the client object making its *call* and the invocation's *arrival* at the service object. Even in a time-sliced, single-CPU machine there may be a time-lag between the *call* and *arrival* events due to a context switch. For these reasons the *call* event in the client object and the *arrival* event in the service object are not synonymous. For similar reasons, the *term* event is not synonymous with the *return* event.

Some other points to note about this event-based model are as follows:

- In this model we assume that events do not overlap. For example, if two invocations arrive at the same time then we assume that their *arrival* events will be ordered (perhaps arbitrarily).
- In the Sos paradigm, *all* invocations upon a service object—including invocations made by the service object on *self*—generate events. A justification of this point is deferred until Section 3.6.1.
- In this thesis, the type of an event is often parameterised with the name of the operation with which the event is associated. For example, *arrival*(Read) denotes an *arrival* event for an operation named *Read*.

3.1.1 Actions

In our event-based model, arbitrary code, referred to as *actions*, can be executed at events. The following notation is used to indicate that an action is to be executed at an event:

event → action

For example:

start(Read) → foo := foo + 1;

This specifies that a variable, *foo*, is to be incremented whenever the event *start*(Read) occurs.

We have already said that events do not overlap. One natural consequence of this is that the execution of an action completes before another event can occur.

3.2 Causing Invocations to Start Executing

The second, core concept of the paradigm is that a way exists to cause an invocation to start executing, i.e., there is a mechanism to trigger the transition from *arrival* to *start* of execution for an invocation. There are several common ways in which synchronisation mechanisms provide this ability:

- One way is for the synchronisation mechanism to provide a statement whose purpose is to initiate execution of the invocation. Examples include the "spawn" and "exec" statements in Mediators [GC86, pg. 470], and the "serve" operations in Eiffel|| [Car90a, pg. 185].
- A variation of this occurs in synchronisation mechanisms that employ locking-type primitives. In such mechanisms, releasing a lock causes an invocation waiting on that lock to continue execution. Examples include condition variables in Monitors [Hoa74] and delay queues in Hybrid [Nie87].
- A third way is to employ guards as in, say, SP [MWBD91]. The guard, associated with an invocation, evaluating to true will trigger the *start* event for that invocation.

The SOS paradigm requires that *some* mechanism be provided to cause invocations to start executing; however, the paradigm does not specify that a *particular* mechanism should be used.

3.3 Access to Information about Invocations

In order to synchronise at the granularity of operation invocations, a synchronisation mechanism needs access to information about these invocations. Furthermore, the greater the information about invocations that a synchronisation mechanism has access to, the more complex the synchronisation policies it is able to implement.

For example, if the *only* information that a synchronisation mechanism has access to is the name of the most recent event in the lifetime of each invocation then about the only useful policy it can implement is mutual exclusion at the granularity of *all* operations. Without access to any other information such as the name of the operation that an invocation is destined for, the synchronisation mechanism does not have the expressive power to implement mutual exclusion at the granularity of individual operations. In other words, for a class that has n operations, such a mechanism can implement the following policy (expressed here in terms of guards and synchronisation counters):

$\mathrm{Op}_1, \mathrm{Op}_2, \ldots, \mathrm{Op}_n$: $exec(\mathrm{Op}_1, \mathrm{Op}_2, \ldots, \mathrm{Op}_n) = 0$;

However, it does not have access to enough information to be able to implement:

Op_1: $exec(\mathrm{Op}_1) = 0$;
Op_2: $exec(\mathrm{Op}_2) = 0$;
...
Op_n: $exec(\mathrm{Op}_n) = 0$;

If the information that a synchronisation mechanism has access to is increased from knowing the most recent event in the lifetime of each invocation to also knowing which operation an invocation is destined for then the synchronisation mechanism can record how often different events occur for each operation—in effect, it has the power of synchronisation counters. This enables it to implement a range of synchronisation policies such as mutual exclusion, several variants of readers/writer, and alternation policy and the bounded buffer.

If a synchronisation mechanism has access to the *arrival* time of invocations then it can implement a FCFS scheduling policy. Other scheduling policies such as the Shortest Job Next scheduler and the Disk Head Scheduler can be implemented if a synchronisation mechanism also has access to invocation parameters.

3.3.1 Information about Invocations Available at Events

The previous section discussed how the expressive power of a synchronisation mechanism is proportional to the amount of information about invocations to which it has access. This section discusses how a lot of information about invocations is available at events. The importance of this is that is shows that event-based synchronisation mechanisms can have good expressive power.

When an invocation *arrives* at a service object, the following information can be known about it:

- The name of the invoked operation.
- The value of any "in" parameters.
- The place where program execution is to resume or results are to be returned when this invocation has completed execution.

Note, however, that while this information will be available, its format might vary wildly from one host language and computer architecture to another. For example, the information stored might be a PC (program counter) return address, the file descriptor of a UNIX-style socket to which results are to be written, the pid (process identifier) of a process to be woken up upon completion, or something else entirely.

- The most recent event associated with this invocation, i.e., *arrival*.

It should also be possible to record another piece of information about the invocation:

- A timestamp (local to the service object) denoting the invocation's *arrival* time.

Similarly, at the *start* event, the same information plus the following can be known about an invocation:

- A timestamp denoting the invocation's *start* time.

Finally, at the *term* event, the same information plus the following can be known about an invocation:

- A timestamp denoting the invocation's *term* time.
- The value of "out" parameters and the operation's "return" value, if any.

It is clear to see that the amount of information about an invocation of an operation that is potentially available at the *arrival*, *start* and *term* events actually exceeds the amount of information about the invocation that is typically available inside the body of the operation. Of course, most of this extra information about invocations that is available at events would be of no use to the code inside the body of an operation (which is why it is not usually made available). However, some of this information *can* be of use to synchronisation mechanisms. For example, the timestamp denoting the relative *arrival* times of invocations could be used to implement a FCFS scheduler.

3.3.2 Accessibility Requirements

The SOS paradigm makes the following accessibility requirements upon invocations:

- All the information about an invocation, that a synchronisation mechanism has access to, should be grouped together as a unit (e.g., as a data structure). This is a modularity requirement.
- An action should have access to information about the invocation for which it is being executed. This is required for expressive power.
- Information about invocations should be maintained in a list/collection over which synchronisation code can iterate in order to compare (and, if need be, update) information about invocations. In particular, information about pending invocations must be accessible. If a synchronisation mechanism so wishes, it may also provide access

to information about invocations that are currently executing or even those that have already terminated execution. Like the previous requirement, this is also for reasons of expressive power.

It is outside the scope of the paradigm to specify any of the following: what representation should be used to store information about an invocation; how information about the *current* invocation is to be accessed; the representation used to store lists/collections of information about invocations; whether lists/collections of invocations should be maintained automatically by the run-time system or manually by programmers; or the language construct used to iterate over such collections. These details will vary from one language to another. However, the principles are illustrated in the following example in which we use a generalised form of the "for loop" [LSAS77] to iterate over invocations:

```
count := 0;
for p in waiting(Print) do
  if p.arr_time < this_inv.arr_time then count ++; endif;
end;
```

Assume that *count* has been declared as a synchronisation variable[1] of the object and that *arr_time* is the time at which an invocation arrived. The **for** loop implicitly declares a variable, *p*, to range over the set of invocations which are currently *waiting* to execute operation *Print*. In executing the above code, *count* records how many of these invocations have arrived before the *current* invocation (denoted by *this_inv*).

Section 4.2 provides some examples to illustrate the utility of being able to access invocations in this manner.

The SOS paradigm places two additional requirements upon accessibility of information about invocations, but these cannot be discussed until after the final core concept of the paradigm has been introduced.

3.4 Separation of Synchronisation Code and Data from Sequential Code and Data

The fourth, and last, component of the SOS paradigm is the requirement that code and data used to synchronise access to an object be segregated from the sequential code and data of the object.

To understand the rationale behind this requirement, consider the diagram in Figure 3.2 in which three types of variable are indicated.

[1] Our paradigm does not specify what syntax should be used to declare synchronisation variables as this will vary from one host language to another. Some example syntax for declaring synchronisation variables can be found in Section 4.2.

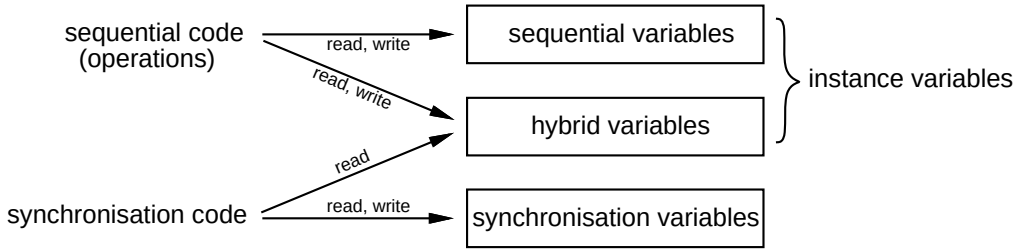

Figure 3.2: Graphical description of an object's code and variables

At the top of the diagram are *sequential* variables of an object, more commonly referred to as instance variables. These variables are used to implement only the sequential functionality of the object.

At the bottom are *synchronisation* variables of the object: variables which are used to implement the synchronisation policy in force on the object, but are not needed to implement the sequential functionality. Two well-known examples are semaphores and synchronisation counters.

In the middle are *hybrid* variables of the object: variables which are *both* sequential (instance) variables and synchronisation variables. Examples of hybrid variables used in examples in Chapter 2 include: (i) *num* in the Bounded Buffer (Figure 2.2 on page 21); (ii) *priority* in the Dynamic Priority Print Queue (Figure 2.3 on page 22); and (iii) *chopstick_avail* in the Dining Philosophers (Figure 2.4 on page 24 and Figure 2.5 on page 25).

Synchronisation code might access these hybrid variables while they are being updated by sequential code and, as has been discussed in Section 2.2, this can result in synchronisation code accessing them while they are in an inconsistent state.

We claim that most hybrid variables are really synchronisation variables which happen to be *implemented* as instance variables (this claim will be supported in Section 4.2.3). All the example synchronisation problems, supposedly requiring access to instance variables, that we have been able to find in the literature have been mis-categorised in this manner. In fact, it is difficult to think of a counter-example.

Thus, if language support is provided for synchronisation variables, and a strict separation of sequential code and data from synchronisation code and data is enforced, then the problems associated with synchronisation code accessing instance variables, as discussed in Section 2.2, disappear.

We defer, until Section 4.5, discussion on how to handle the hypothetical case of variables which are truly hybrid.

3.5 Accessibility Requirements Revisited

Having discussed the requirement of separation between sequential code/data and synchronisation code/data, we can now finish the discussion about access to information about

invocations.

Section 3.3.2 listed three accessibility requirements on invocations that the Sos paradigm makes. There are two more:

- We previously discussed how it would be dangerous for synchronisation code to access instance variables due to the possibility of the variables being examined while in an inconsistent state. Similarly, it would be dangerous for synchronisation code to access parameters if there was a possibility of them being updated while the synchronisation code was examining them. This issue needs to be resolved if synchronisation code is to have safe access to parameters. (This is discussed in Section 3.5.1.)
- It should be possible for programmers to declare synchronisation variables local to invocations. (This is discussed in Section 3.5.2.)

In making these two final requirements, we achieve the symmetry of sequential and synchronisation variables shown in Table 3.1. Thus, we argue, the introduction of "synchronisation variables" has not added new concepts to language design, but rather has generalised the existing concept of "variables."

Variable Type	Sequential	Synchronisation
variables of an object	supported	supported
parameters	supported	supported
local variables	supported	supported

Table 3.1: Symmetry of sequential and synchronisation variables

3.5.1 Parameters

Parameters are most often used to help implement the sequential functionality of an operation. We refer to these as *sequential* parameters.

Sometimes a parameter is used to implement a scheduling policy, and is not actually being used in the sequential body of an operation. A well-known example of this is the Shortest Job Next scheduler [BH78] in which a parameter, *len* (indicating the estimated length of the job), is used to schedule invocations. In this case *len* is said to be a *synchronisation* parameter. Other examples of synchronisation parameters include *gid* (the group identifier) used in the Dynamic Priority Print Queue (Figure 2.3 on page 22), and the table position, *pos*, passed to *Eat* in the Dining Philosophers problem (Figure 2.4 on page 24 and Figure 2.5 on page 25).

Although hybrid variables of an object rarely, if ever, occur, hybrid parameters are more common. An example can be found in the Disk Head Scheduler [Hoa74]: the parameter indicating the part of the disk to which data is to be transferred is used in both the sequential code (to physically move the disk head) and the synchronisation code (to schedule invocations in order to minimise movement of the disk head).

The SOS paradigm stipulates that hybrid parameters must not be accessed by synchronisation code while they are being updated. Some possible techniques to ensure this include:

- The run-time system could arrange for the parameters of an invocation to be copied when the invocation arrives, i.e., at the *arrival* event. These parameter *copies* could be accessed by the synchronisation code, safe in the knowledge that if the sequential body of an operation updated a parameter then this update would not affect the synchronisation code's copy of that parameter (and vice versa). A simple optimisation is for the run-time system to copy only those parameters which are used in synchronisation code.
- The host language might make a restriction that the only type of parameters that can be accessed by both sequential code and synchronisation code are those that are declared to be read-only (akin to "in" parameters in Ada).

Although SOS requires that *some* technique be used, it does not specify *which*. This is because the most appropriate technique to use will vary from one host language to another.

3.5.2 Local Variables

The concept of local variables is well-known in sequential programming languages. The concept also makes sense when discussing synchronisation mechanisms. We are not suggesting that variables local to the sequential body of an operation should be accessible to synchronisation code: such an arrangement would lead to problems similar to those involved in permitting synchronisation code to access instance variables. Rather, we propose that there should be two categories of local variables: *sequential* local variables (what are commonly referred to as "local variables") and *synchronisation* local variables.

Note that sequential local variables and synchronisation local variables have different lifetimes. The sequential local variables come into existence at the *start* of execution of an invocation and are discarded at the *termination* of execution. Synchronisation local variables come into existence at the *arrival* of an invocation. They are usually discarded at the *termination* of execution. However, the SOS paradigm does not preclude the possibility of information about invocations being kept after the *termination* of execution so it is possible that synchronisation local variables might outlive the *termination* of an invocation's execution.

A well-known example of a synchronisation local variable is a timestamp associated with each invocation to record its arrival time; this can be used to implement a FCFS scheduler.

Another example appears in a variation of the Shortest Job Next scheduler. Rather than have the estimated job length passed in as a parameter, it might be possible to arrange for the synchronisation code to calculate the value itself and record this information in a synchronisation variable local to the relevant invocation.[2]

[2]There are two advantages to having the synchronisation code calculate the length of submitted jobs rather than have clients pass the length in as a parameter. Firstly, it moves the inconvenience of calculating job

3.6 Discussion

Having introduced the core concepts of the paradigm, we now take the opportunity to tidy up a few loose ends and make some comments on the overall paradigm.

3.6.1 Invocations upon *Self*

This section offers a justification for the decision, mentioned in passing in Section 3.1, that *all* invocations upon a service object, including *self* invocations, generate events.

In the Sos paradigm, synchronisation occurs at *arrival*, *start* and *term* events which mark the lifespan of invocations. If a particular invocation was exempt from generating such events then it would be impossible to synchronise it. Thus, in stating that *all* invocations generate events, we are stating that *all* invocations can be synchronised. We have no objection to a compiler optimising away events that it has determined are not used.

An alternative, as practiced in some languages (e.g., CEiffel [Löh93] and more recently suggested for Guide [Riv92, RR92], is that invocations which originate from *other* objects are subject to synchronisation but invocations that originate from the service object itself, i.e., invocations on *self*, are exempt from synchronisation. This has the effect of making *self* invocations semantically different from other invocations. In general, it is a good idea to avoid introducing idiosyncrasies into a language without good reason and there appears to be no good reason for introducing this particular one.

There are two arguments that might be used for advocating treating the synchronisation of *self* calls differently to that of non-self calls. However, both are flawed, as we now discuss.

The first argument is that such treatment facilitates the writing of classes in which some operations are implemented in terms of other operations. For example, consider a bounded buffer class that, along with *Put* and *Get* operations, also includes *Get2*, which obtains two consecutive items from the buffer. It might seem natural to implement *Get*2 by having it invoke *self.Get* twice. However, if these operations execute in mutual exclusion then doing so would result in deadlock and hence, it is argued, *self* calls should not be synchronised.

However, in such cases, the problem can be easily solved by introducing an extra, private operation to the class, say, *ActualGet*. *Get* is implemented by having it invoke *self.ActualGet* once and *Get*2 is implemented by having it invoke *self.ActualGet* twice. Suitable synchronisation constraints (expressed here as guards) for these operations would then be of the form:

Put: *exec*(Put, Get, Get2) = 0 **and** ... ;
Get: *exec*(Put, Get, Get2) = 0 **and** ... ;
Get2: *exec*(Put, Get, Get2) = 0 **and** ... ;
ActualGet: **true**;

length from clients to the service object, thus making the service more convenient to use. Secondly, it prevents clients from thwarting the service object's scheduling policy by deliberately under-estimating the length of their own jobs.

The second argument is based on the mistaken belief that in a sequential system, code runs in mutual exclusion. If this belief is accepted then, it is reasoned, in order to be able to reuse existing object-oriented, sequential code safely in a concurrent environment, objects should, by default, be controlled by a mutual exclusion synchronisation policy [Löh93] [Pap93, ch. 4]. If this argument is accepted then it follows that calls upon *self* would need to be unsynchronised in order to prevent deadlock (as reasoned in the first argument above).

This argument implicitly assumes the granularity of synchronisation to be at the level of objects. This assumption does not hold for the Sos paradigm which, as stated in Section 3.1, provides synchronisation at the granularity of operation invocations. Hence, even if the argument were valid, it would not be relevant to the Sos paradigm.

However, the argument is not valid since it is based upon the mistaken belief that in a sequential system code runs in mutual exclusion. If this were the case then sequential programs that contained recursive calls would deadlock. Rather, sequential programs are "dangerous" in that they do not contain *any* synchronisation constraints, e.g., mutual exclusion, to prevent data corruption and it is only the fact that sequential programs do not contain multiple threads that prevents data corruption.

Rather than writing classes with the assumption that they will be used in a sequential environment and then trying to reuse them in a concurrent environment by introducing language idiosyncrasies, it might be better to write classes with the assumption that they will be used in a concurrent environment. This would guarantee the ability to safely reuse a class in a sequential environment since a sequential program is, in effect, a special case of a concurrent program that happens to contain just one thread. However, further discussion of this topic is outside the scope of this thesis.

3.6.2 A Single Source of Information

In Section 2.1 we mentioned that Bloom [Blo79] claims a synchronisation mechanism should have access to six different types of information in order to have good expressive power. In actual fact, all of these six types of information either are information about invocations or information that is derived from invocations.

For instance, three of Bloom's types of information are: the relative arrival time of invocations, invocation parameters and the name of the operation upon which an invocation was made. As we illustrated in Section 3.3.1, these three pieces of information about an invocation are known at the invocation's *arrival*.

The other three types of information that Bloom lists are: the synchronisation state of the object, history information and instance variables. We claimed in Section 3.4 (and we will support this claim in the next chapter) that a synchronisation mechanism does not, in fact, need access to the instance variables of an object. Rather it can maintain the information it needs itself in the form of synchronisation variables. Bloom's synchronisation state and history information can be provided by synchronisation counters which, as we show in Section 4.2.1, are also synchronisation variables. Synchronisation variables are maintained at

events. Since events are associated with invocations it follows that synchronisation variables denote information that is derived from invocations.

Thus we see that a synchronisation mechanism requires access to just one primary source of information, not six derivative sources as Bloom claims, in order to be expressively powerful.

3.6.3 Suitability for Different Computational Models

Concurrent, object-oriented languages can differ greatly in important aspects. For instance, some languages support concurrency *within* objects while others support concurrency only *between* objects. Some languages embody a shared-memory paradigm and operation invocations are based on procedure calls—transparently utilising RPC (remote procedure call) in the event that an object being invoked is on a different node. Other languages do not assume the existence of shared memory and implement operation invocations by message-passing.

Such key considerations in language design can have an effect on the design of synchronisation mechanisms. It is quite common for a synchronisation mechanism to be designed for a particular language. In such cases one might wonder if the synchronisation mechanism is dependent upon some characteristic aspects of the host language, or if it might be possible to use that synchronisation mechanism in other concurrent, object-oriented languages.

In designing the Sos paradigm, we have striven to avoid making unportable assumptions about the underlying object model. In particular, throughout this chapter we have used the term *invocation* without ever making any assumptions as to how an invocation is performed, e.g., by procedure call or message-passing. Similarly, we have not made any assumptions as to whether concurrency within an object is supported or if, instead, the servicing of invocations will be serialised.

In avoiding such assumptions, the principles of the Sos paradigm should be applicable to a wide range of concurrent, object-oriented languages.

3.7 Summary

This chapter has discussed the core concepts that constitute the Sos paradigm. The paradigm will be used throughout the rest of this thesis, starting in the next chapter when we develop a sample synchronisation mechanism to illustrate the paradigm. Throughout the thesis, it will be shown that the Sos paradigm offers several benefits, including:

- We have argued that a synchronisation mechanism requires access to just one primary source of information, rather than six derivative sources as Bloom claims [Blo79], in order to have good expressive power. Since the Sos paradigm supports this primary source of information, Sos-based synchronisation mechanisms can be expressively powerful. Furthermore, minimising the number of sources of information can keep low the number of constructs employed by a synchronisation mechanism and hence Sos-

based synchronisation mechanisms can achieve their expressive power without the risk of complexity or creeping featurism.

- By providing synchronisation code with the means to maintain its own variables, synchronisation code no longer needs to access the instance variables of an object. As such, the problems of unsafe access to instance variables are avoided.
- We will show that most of the concepts of the Sos paradigm can be implemented in existing language constructs, thus permitting a Sos-based synchronisation mechanism to be added to a host language with relative ease.
- Sos offers modularity by completely separating synchronisation code and variables from sequential code and variables. We will shown in Part III that this separation facilitates practical language support for generic synchronisation policies, and in Part IV that it helps reduce the harmful effects of the ISVIS conflict,

In drawing this chapter to a close, it would be foolish to claim that the Sos paradigm is a panacea for all the ills of concurrent programming. For instance, the paradigm is concerned *solely* with synchronisation in service objects and does not address synchronisation issues in client objects. Even within the domain of service objects, there are several issues that the paradigm does not address, though we feel it could be extended to do so. A discussion of these issues can be found in the "Future Work" chapter (Chapter 14).

Chapter 4

Esp—A Sample Synchronisation Mechanism to Illustrate the Sos Paradigm

In this chapter we illustrate the Sos paradigm by means of a sample synchronisation mechanism. At first sight, the mechanism looks like it is an enhancement of SP [MWBD91] and so, for want of a better name, it was christened Esp (an Extension of Scheduling Predicates).

The structure of this chapter is as follows. Section 4.1 briefly explains the pseudo-code notation that is used throughout this chapter and, indeed, much of the rest of this thesis. Then, in Section 4.2, an extensive series of examples is used to introduce, and illustrate the power of, the Esp synchronisation mechanism. In Section 4.3 we discuss how Esp is both a declarative and a procedural synchronisation mechanism. Unfortunately, if care is not taken with implementation then the price paid for Esp's expressive power might be poor run-time performance. With this in mind, Section 4.4 outlines some optimisation techniques that might be employed. Section 4.5 returns to the topic of hybrid variables which we discussed in the previous chapter. The chapter is brought to a close in Section 4.6 which summaries the main points contained herein.

4.1 Notation

Figure 4.1 shows the layout of a class in Esp; the lines are numbered for ease of reference. Note that a class is split up into a sequential part and a synchronisation part with the **synchronisation** keyword (line 5) separating the two. The symmetry of the class's components is an attractive feature (which is enhanced by the comment on line 2). The sequential code may not invoke any operations in the synchronisation code or access any of its variables, and vice versa.

The code to update synchronisation variables is placed in *actions* (line 8). If the code in an action becomes large or is replicated then the programmer may wish to place some of it

```
 1: class Foo {
 2: // sequential
 3:    variables
 4:    operations
 5: synchronisation
 6:    variables
 7:    operations
 8:    events → actions
 9:    guards
10: }
```

Figure 4.1: Layout of a class in Esp

into synchronisation operations (line 7) which the actions can invoke.

4.1.1 Guard Evaluation Semantics

The semantics of the guards employed by Esp are intuitive: an invocation will be permitted to *start* execution *as soon as* its guard evaluates to true. One may think that in order to guarantee such *as soon as* semantics, an implementation would have to enter a loop to continually evaluate guards. However, this is not the case.

The strict segregation of synchronisation code/variables from sequential code/variables that Sos imposes means that the only variables which a guard can reference are synchronisation variables. Furthermore, from the event-based nature of Sos, we know that these synchronisation variables can be updated only at events. Thus, assuming that guards are free from side effects, if a guard evaluates to false then the earliest occasion at which it might re-evaluate to true is at the next event. As such, re-evaluating guards at every event is sufficient to guarantee that guards will evaluate to true (and hence an invocation will *start* execution) as soon as possible.

4.2 Examples

We now illustrate the usage of synchronisation variables through examples. These examples are organised as follows. The examples in Section 4.2.1 illustrate how several synchronisation mechanisms are subsumed by Esp, and hence by the Sos paradigm. The examples in Section 4.2.2 illustrate how complex scheduling policies can be implemented easily. Finally, the examples in Section 4.2.3 are those commonly found in the literature that traditionally have been implemented with the aid of instance variables; we show how these can be implemented with synchronisation variables.

4.2.1 Subsumption of Other Synchronisation Mechanisms

The examples in this section show how Esp subsumes several synchronisation mechanisms.

Synchronisation Counters

The code in Figure 4.2 declares three synchronisation variables—a, b and c. These are all initialised to zero when an object is created. (In the pseudo-code notation we are using, the constructor of a class is an operation with the same name as the class itself. Thus *start*(Foo) is an event associated with the start of execution of the constructor of an object of type *Foo*.) Actions to increment a, b and c are executed whenever the events *arrival*(Bar), *start*(Bar) and *term*(Bar), respectively, occur. Two of these variables are then used in the guard on *Bar*, which implements mutual exclusion.

```
class Foo {
  Foo(...) { ... }      // constructor
  Bar(...) { ... }
synchronisation
  int   a, b, c;
  start(Foo) → { a := 0; b := 0; c := 0; }
  arrival(Bar) → a ++;
  start(Bar) → b ++;
  term(Bar) → c ++;

  Bar: b − c = 0;
}
```

Figure 4.2: Implementing synchronisation counters

In effect, this example uses synchronisation variables to implement synchronisation counters. (The variable a implements the counter *arrival*(Bar), and so on.) Thus we see that synchronisation variables subsume the power of synchronisation counters. Although subsumed, synchronisation counters are provided in Esp as useful syntactic sugar.

An interesting point is that Scheduling Predicates, upon which Esp is based, also subsumes the power of synchronisation counters, but in a different manner [MWBD91, pg. 187].

Relative Arrival Time of Invocations

The previous example illustrates how synchronisation variables of an object can be declared —the syntax is similar to that used to declare *sequential* variables of an object (i.e., instance variables).

The next example (Figure 4.3) shows how to declare a synchronisation variable, *arr_time*, local to (invocations of) an operation. Each invocation for operations A, B and C is given its

own instance of *arr_time*. Whenever an invocation arrives (denoted by an *arrival* event), the *arr_time* variable of that invocation will be assigned the current value of the *clk* variable and *clk* will be incremented. In this way, each invocation will have a unique value for *arr_time*. The expression *this_inv.arr_time* can then be used in guards to schedule invocations based on their relative arrival time.

```
class Foo {
   Foo(...) { ... }      // constructor
   A(...) { ... }
   B(...) { ... }
   C(...) { ... }
synchronisation
   int    clk;
   int    arr_time local to A, B, C;

   start(Foo) → clk := 0;
   arrival(A, B, C) → this_inv.arr_time := clk ++;
   ⋮
}
```

Figure 4.3: Implementing the relative *arrival time* of invocations

Several synchronisation mechanisms (e.g., Scheduling Predicates, Eiffel|| [Car90a] and CEiffel [Löh91]) provide programmers with access to the relative arrival time of invocations. The above example illustrates that, as for synchronisation counters, this functionality is simply a form of syntactic sugar for synchronisation variables.

As with synchronisation counters, ESP automatically maintains *arr_time* for the convenience of programmers.

Scheduling Predicates

Consider the following guard which implements a FCFS queue:

```
Print: exec(Print) = 0
       and there_is_no(p in waiting(Print): p.arr_time < this_inv.arr_time);
```

The code in Figure 4.4 implements the same functionality using a **for** loop to iterate over invocations. The **if** statement's condition (in the body of the loop) was derived directly from the condition used in the *there_is_no* predicate above. Thus we see that ESP subsumes Scheduling Predicates. As with synchronisation counters and *arr_time*, ESP provides scheduling predicates as a form of syntactic sugar.

```
class FCFSPrinter {
  Print(...) { ... }
synchronisation
  Boolean NoInvocationBeforeMe(int myTime)
  { for p in waiting(Print) do
      if p.arr_time < myTime then
        return false;
      endif;
    end;
    return true;
  }
  Print: exec(Print) = 0 and NoInvocationBeforeMe(this_inv.arr_time);
}
```

Figure 4.4: First-come, first-served Printer

Path Expressions and the "by" Statement of SR

Path Expressions can be implemented in terms of synchronisation counters [McH89] [Cam83]. Since Esp subsumes the power of synchronisation counters, it follows that Esp also subsumes the power of Path Expressions.

The synchronisation mechanism of the SR language [And81] provides a **by** statement which is used to schedule invocations of an operation based on a parameter. For example, assuming operation *Print* takes a parameter called *len*, the following guard implements a Shortest Job Next scheduler:

Print: *exec*(Print) = 0 **by** len;

This guard containing a **by** clause is equivalent to the following guard containing a scheduling predicate:

Print: *exec*(Print) = 0 **and** *there_is_no*(p **in** *waiting*(Print): p.len < *this_inv*.len);

Thus we see that scheduling predicates subsume the power of the **by** clause. Since Esp subsumes the power of scheduling predicates, it follows that Esp also subsumes the power of the **by** clause.

4.2.2 Scheduling Power

The following examples show how Esp can implement various complex scheduling policies.

Alarm Clock

In the Alarm Clock problem [Hoa74], an operation, *WakeUp*, must be implemented which will delay for a specified *period* of time. A usual assumption is that it is possible to arrange

for another operation, *Tick*, to be invoked periodically (say, once a second) to mark the passage of time.

Our solution is shown in Figure 4.5. In this, the counter *term*(Tick) is used to indicate the current time. We associate a variable, *wakeup_time*, with each *WakeUp* invocation and calculate its value at the *arrival*(WakeUp) event. The resultant guard on *WakeUp* is trivial and intuitive.

```
class AlarmClock {
  WakeUp(int period) { }
  Tick() { }
synchronisation
  int    wakeup_time local to WakeUp;
  arrival(WakeUp) → this_inv.wakeup_time := term(Tick) + this_inv.period;
  WakeUp: term(Tick) >= this_inv.wakeup_time;
}
```

Figure 4.5: The Alarm Clock

Notice that the operations *WakeUp* and *Tick* have empty bodies. This is because the Alarm Clock is purely a synchronisation problem. The operations are, in effect, just hooks into the synchronisation code. Being able to view the Alarm Clock as a purely synchronisation problem is not possible in many other languages in which synchonisation code has to be combined with the sequential code of operations in order to implement it. The Sina [TA88, pg. 32–33] and Monitor [Hoa74, pg. 553–554] implementations are prime examples.

Shortest Job Next (Starvation-free Version)

In the Shortest Job Next scheduler [BH78], jobs are serviced in reverse order of their estimated length. Thus it is possible for a long job to be skipped over indefinitely by a continuous stream of shorter jobs. One way to overcome this inherent unfairness is to adjust the priority (in this case, the estimated length) of jobs which are skipped over so that they are less likely to be skipped over in future.

Figure 4.6 shows our solution to this. The guard on *Print* implements the basic scheduler, and the action associated with the *start*(Print) event iterates through all of the *waiting* invocations to decrement the *len* variable of any that have been skipped over.

If it is preferred that clients pass in their job's estimated length as a parameter rather than have it calculated locally then this can be easily accomodated by removing the declaration of *len* as a synchronisation variable and instead declaring it as a parameter to *Print*.[1]

[1] This requires, as discussed in Section 3.5.1 on page 40, that the run-time system arranges for two copies of parameters to be maintained: one for the sequential code and the other for the synchronisation code. Thus decrementing *len* would only affect the synchronisation code's copy of this parameter.

```
class FairSJN {
  Print(string fileName) { ... }
synchronisation
  int   len local to Print;
  arrival(Print) → this_inv.len := ... // use a system call to determine file length
  start(Print) →
  { for p in waiting(Print) do
      if p.arr_time < this_inv.arr_time then p.len --; endif;
    end;
  }
  Print: exec(Print) = 0 and there_is_no(p in waiting(Print): p.len < this_inv.len or
                                  p.len = this_inv.len and p.arr_time < this_inv.arr_time);
}
```

Figure 4.6: Starvation-free, Shortest Job Next scheduler

It is equally trivial to obtain the basic (unfair) SJN scheduler—just delete the action associated with the *start*(Print) event.

Dining Philosophers (Starvation-free Version)

The Guide solution to the Dining Philosophers problem shown in Figure 2.5 on page 25 does not guarantee to prevent starvation of a philosopher by conspiracy on the part of her immediate neighbours to keep her blocked. One way to prevent such starvation is to set an upper limit on how many times a philosopher may be skipped over.

The code in Figure 4.7 illustrates this approach. This solution makes use of two predicates in the guard of *Eat*.

The first predicate examines parameters of the currently *executing* invocations to determine if it is possible for a philosopher to start eating.

The purpose of the second predicate is to prevent a waiting philosopher being skipped over indefinitely (no more than three times is the threshold value used). This predicate relies on a variable, *skipped*, being maintained for each pending *Eat* invocation. When an *Eat* invocation arrives, its *skipped* variable is initialised to zero; whenever a philosopher is allowed to eat—designated by a *start*(Eat) event—a **for** loop determines which invocations were skipped over, and increments their *skipped* values.

4.2.3 Instance Variables

In Section 3.4, we claimed that if a language provides support for synchronisation variables then the synchronisation code of an object will not require access to any variables used by the sequential code. We now substantiate this claim by taking several synchronisation problems

```
type TablePositon is int subrange(0..4);
class FairDiningPhilTable{
  Eat(TablePosition pos) { ... }
synchronisation
  int skipped    local to Eat;
  arrival(Eat) → this_inv.skipped := 0;
  start(Eat) →
  { for p in waiting(Eat) do
      if ShareForks(p.pos, this_inv.pos) and p.arr_time < this_inv.arr_time
        then p.skipped ++; endif;
    end;
  }
  boolean ShareForks(TablePosition i, TablePosition j)
  // we "share forks" with somebody if they are sitting at
  // our position, to our immediate left or immediate right
  { return (i + 1) mod 5 = j or i = j or (j + 1) mod 5 = i; }

  Eat: there_is_no(p in executing(Eat): ShareForks(p.pos, this_inv.pos)) and
       // the rest of this guard is to prevent starvation
       there_is_no(p in waiting(Eat): ShareForks(p.pos, this_inv.pos) and
                                        p.arr_time < this_inv.arr_time and p.skipped >= 3);
}
```

Figure 4.7: Starvation-free solution to the Dining Philosophers problem

from the literature that have been implemented using instance variables, and re-implement them using synchronisation variables instead.

Dynamic Priority Print Queue

In Section 2.2.1, we introduced the Dynamic Priority Printer and discussed the problems associated with trying to maintain the group priorities as instance variables. Figure 4.8 shows an Esp implementation of this scheduler. The main difference between this and the previous attempt (Figure 2.3 on page 22) is that the *priority[]* variable and the code to maintain it have been moved from the sequential part of the object to the synchronisation part. A minor side-effect is that the *UpdatePriority* operation now has an empty body since (like the *WakeUp* and *Tick* operations in the Alarm Clock) it is, in effect, just a hook into the synchronisation code.

This movement of variables and the code to maintain them into the synchronisation part of an object does *not* result in a decrease in code size; indeed the amount of code is the same as before. Rather, the benefits we gain are that: (i) the code is more modular, since

```
type GroupId = (one, two, three, four, grad, staff);

class Printer {
  Printer(...) { ... }      // constructor
  UpdatePriority(GroupId gid, int NewPriority) { }
  Print(GroupId gid, string FileName) { ... }
synchronisation
  int   priority[GroupId];

  start(Printer) → { "initialise priority[]"; }
  start(UpdatePriority) → { priority[this_inv.gid] := this_inv.NewPriority; }
  Print: exec(Print) = 0 and there_is_no(p in waiting(Print):
       priority[p.gid] > priority[this_inv.gid] or
       priority[p.gid] = priority[this_inv.gid] and p.arr_time < this_inv.arr_time);
}
```

Figure 4.8: Solution to the Dynamic Priority Print Queue

all variables/code to implement the scheduling policy of the object are separated from those which implement its functionality (in this case, the ability to print a file); and (ii) it is easier to ensure the synchronisation code is correct with *priority[]* as a synchronisation variable rather than a sequential variable.

The Bounded Buffer

A bounded buffer implemented with a fixed size array needs two variables for its maintenance: *put_index* specifies the array position at which *Put* should place the next item, and *get_index* specifies the array position at which *Get* should retrieve the next item.

A third variable, *num*, is used to check for underflow and overflow. This latter variable is a synchronisation variable, while *put_index* and *get_index* are sequential (instance) variables.

Some implementations of bounded buffer in the literature drop the variable *put_index* since its value can be calculated by the formula:

$$put\_index = (get\_index + num) \textbf{ mod } \text{“size of array”}$$

However, this results in *num* being used in both the sequential code as well as the synchronisation code (as previously shown in Figure 2.2 on page 21), thus making it a hybrid variable. Having the sequential code maintain both *put_index* and *get_index* and leaving *num* as a synchronisation variable retains modularity. It also brings the benefit of increased potential for concurrency: since *Put* manipulates only *put_index* and *Get* manipulates only *get_index*, *Put* and *Get* can execute concurrently with each other.

The implementation of the bounded buffer shown in Figure 4.9 does not explicitly declare

```
class Buffer[elem, Size] {
  int   get_index, put_index;          elem   data[Size];
  Buffer() { get_index := 0;   put_index := 0; }
  Put(...) { ... "update put_index"; ... }
  Get(...) { ... "update get_index"; ... }
synchronisation
#define num      term(Put) − term(Get)
  Put: exec(Put) = 0 and num < Size;
  Get: exec(Get) = 0 and num > 0;
}
```

Figure 4.9: ESP solution to the Bounded Buffer

num. Rather, this implementation relies on the fact that *num* is incremented for every *Put* and decremented for every *Get*. Thus, its value is given by the formula:

$$num = term(\text{Put}) - term(\text{Get})$$

Dining Philosophers (Revisited)

Figure 4.7 on page 53 showed an implementation of the Dining Philosophers that made use of predicates. Now we show an alternative solution—one that makes use of synchronisation variables of the object rather than relying on predicates.

The code in Figure 4.10 shows this solution. An array of booleans, *chopstick_avail[]*, indicates the availability of each chopstick at the table. Initially all chopsticks are available. When a philosopher starts to eat (indicated by the *start*(Eat) event), *chopstick_avail[]* is updated to indicate that the appropriate chopsticks are now in use. Similarly, when a philosopher finishes eating, the chopsticks are once again marked as being available. With the availability of the chopsticks being maintained at events, the guard on *Eat* is trivial and reflects the problem description. This can be compared with the Guide solution (Figure 2.5 on page 25).

The code in this example shows actions invoking synchronisation operations. We could have written the action code "inline" but we feel that the use of synchronisation operations improves the clarity of the code.

Disk Head Scheduler

Several different algorithms exist to schedule the transfer of data to/from a disk. For example, to minimise head movement, a "nearest job next" policy might be used [And81, pg. 418–419]. However, this could result in starvation of invocations that are far away from the disk head. An alternative strategy is to serve invocations in one direction until there are no more and

```
type TablePositon is int subrange(0..4);
class DiningPhilTable {
  DiningPhilTable(...) { ... }      // constructor
  Eat(TablePositon pos) { ... }
synchronisation
  boolean    chopstick_avail[TablePositon];

  init()    { "set all chopstick_avail[] to true"; }
  toggle_chopsticks(TablePosition pos)
  { chopstick_avail[pos] := not( chopstick_avail[pos] );
    chopstick_avail[(pos + 1) mod 5] := not( chopstick_avail[(pos + 1) mod 5] );
  }
  start(DiningPhilTable) → init();
  start(Eat), term(Eat) → toggle_chopstick(this_inv.pos);

  Eat: chopstick_avail[this_inv.pos] and chopstick_avail[(this_inv.pos + 1) mod 5];
}
```

Figure 4.10: Solution to the Dining Philosophers problem

then reverse direction [Hoa74]. This is sometimes called the "elevator algorithm" because it mimics the behaviour of a lift.

The code in Figure 4.11 implements the elevator algorithm. The *Distance* function in the synchronisation part of the class calculates the maximum distance the disk head might have to travel to get to an invocation. The body of this function is somewhat complex as the calculation is dependent not only on the relative positions of the disk head and the invocation, but also on the current direction of travel of the disk head. If a "nearest job next" policy was desired then the body of *Distance* would be simpler; in fact, it would contain just a single statement:

```
return abs(headPos − dest);
```

The synchronisation code updates the position of the disk head, *headPos*, and its direction of travel, *goingUp*, whenever a call to *Transfer* is permitted to *start* execution. With this infrastructure in place the guard on *Transfer* is trivial and intuitive.

The code in Figure 4.11 provides a single operation, *Transfer*, for which invocations are scheduled and there is no way for a client of this class to bypass the scheduler. This is in contrast with the Monitor implementation of the Disk Head Scheduler [Hoa74, pg. 555–556] which relies on clients to obey the following protocol:

```
diskhead.Request
  client code to transfer the data
diskhead.Release
```

```
const MaxCylinder = 100;
type CylinderNum is int subrange(0..MaxCylinder);
class DiskHeadScheduler {
  DiskHeadScheduler(...) { ... }      // constructor
  Transfer(CylinderNum dest, DataBlock data)
  {
    "move the disk head to 'dest'"
    "transfer the 'data' to the disk"
  }
synchronisation
  CylinderNum   headPos;          boolean   goingUp;

  int Distance(CylinderNum dest)
  {
    if dest = headPos then
      return 0;
    elsif goingUp and dest > headPos or not(goingUp) and dest < headPos then
      return abs(headPos - dest);
    elsif goingUp and dest < headPos then
      return 2 * MaxCylinder - headPos - dest;
    elsif not(goingUp) and dest > headPos then
      return headPos + dest;
    endif
  }

  MaintainDirection(CylinderNum dest)
  {
    if dest < headPos then
      goingUp :=false;
    elsif dest > headPos then
      goingUp :=true;
    endif;
  }

  start(DiskHeadScheduler) → { headPos := 0; goingUp := true; }
  start(Transfer) →
  {
    MaintainDirection(this_inv.dest);
    headPos := this_inv.dest;
  }

  Transfer: exec(Transfer) = 0 and there_is_no(t in waiting(Transfer):
                                        Distance(t.dest) < Distance(this_inv.dest));
}
```

Figure 4.11: First solution to the Disk Head Scheduler

(The *Request* and *Release* operations are provided by the monitor but *Transfer* is not.) It would be possible for clients of such a monitor to disregard the protocol and transfer data in an unsynchronised manner.

A criticism of our implementation of the Disk Head Scheduler is that repeatedly invoking *Distance* within the guard is inefficient. We can combat this inefficiency as follows.

In the elevator algorithm, the distance of a pending invocation from the disk head—as calculated by the *Distance* function—is highest when the invocation first arrives and decreases thereafter. It decreases whenever any other invocation is serviced and it is possible to calculate by how much it decreases. Thus, one could declare and maintain a variable, *distance*, local to each *Transfer* invocation and use this variable in the guard in place of invoking *Distance*. Figure 4.12 illustrates the changes in code needed to achieve this.

```
const MaxCylinder = 100;
type CylinderNum is int subrange(0..MaxCylinder);
class DiskHeadScheduler {
  DiskHeadScheduler(...) { ... }      // constructor
  Transfer(CylinderNum dest, DataBlock data) ...
synchronisation
  int    distance local to Transfer;
  ... // unchanged code deleted to save space
  MaintainInvocationDistance(int deltaHeadMovement)
  {
    for t in waiting(Transfer) do
      t.distance := t.distance - deltaHeadMovement;
    end;
  }

  arrival(Transfer) → { this_inv.distance := Distance(this_inv.dest); }
  start(Transfer) →
  {
    MaintainInvocationDistance( Distance(this_inv.dest) );
    MaintainDirection(this_inv.dest);
    headPos := this_inv.dest;
  }

  Transfer: exec(Transfer) = 0 and there_is_no(t in waiting(Transfer):
                                                  t.distance < this_inv.distance);
}
```

Figure 4.12: Optimised solution to the Disk Head Scheduler

This code is still inefficient, albeit not as inefficient as the first solution we presented. However, further improvements may still be possible with the aid of an optimising compiler. Later, in Section 4.4, we mention some compile-time optimisations. One of particular relevance to the current example is optimisation by transformation. Briefly, if a compiler can recognise that a certain pattern of guards specifies a particular synchronisation policy then it

could generate code to implement that policy in a more efficient manner. For example, if the compiler recognises that a guard schedules invocations based on the value of a parameter, or a synchronisation local variable, then it could generate code to maintain an ordered list of invocations; whenever an invocation is to be allowed to start executing, the run-time system need only choose the invocation at the head of the list.

The guard for *Transfer* in Figure 4.12 is in this optimisable form: it schedules invocations based on the *distance* variable of each invocation. Thus an optimising compiler could generate code to maintain a list of *Transfer* invocations that is ordered by *distance*. The only complication is that the action associated with the *start*(Transfer) event updates the *distance* variable of pending invocations. In this particular example *distance* is decremented by the same amount for each invocation and hence there is no change in the relative order of invocations. However, in general, such modifications might necessitate a re-sort of the list.

Table 4.1 compares the run-time cost of our two versions of the Disk Head Scheduler. In our first solution (Figure 4.11), the run-time system can simply append newly arrived invocations to the end of the invocation list, taking O(1) time, but then it potentially has to make $O(N^2)$ comparisons when evaluating the guard of *Transfer* in order to decide which invocation should execute next. If a compiler optimises our second solution then newly arrived invocations will require O(N) comparisons to place them into the invocation list and choosing the next invocation to be executed is simply a matter of removing the head element of the list, taking O(1) time. This is as fast as the Monitor's solution [Hoa74, pg. 555–556] which also requires O(N) time for insertion into a condition queue and O(1) time for removal. Thus we see that the high-level nature of Esp need not result in poor performance.

	insert item	remove item	total cost
Original code (Figure 4.11)	O(1)	$O(N^2)$	$O(N^2)$
Optimised code (Figure 4.12)	O(N)	O(1)	O(N)

Table 4.1: Cost of maintaining the invocation list for the Disk Head Scheduler

4.3 Esp is both Declarative *and* Procedural

If programmers use only guards and the automatically maintained synchronisation variables (synchronisation counters, the *arr_time* and parameters of invocations) of Esp then they use it in a purely declarative manner. This declarative subset of Esp is, in effect, SP and it has good expressive power. When faced with a synchronisation policy that is outside the expressive power of SP, programmers can use actions to maintain extra synchronisation variables. As more and more use is made of actions, the more procedural (and, hence, less declarative) Esp appears to be. Thus one might consider Esp to be a declarative mechanism that degrades gracefully into a procedural mechanism.

On the other hand, we showed in Section 4.2.1 that most of the declarative constructs

of Esp—synchronisation counters, the relative arrival time of invocations, and scheduling predicates—are simply forms of syntactic sugar. If these syntactic sugar constructs were removed then the remaining core of Esp would be quite procedural in nature. Thus, one might consider Esp to be a procedural mechanism that happens to provide some declarative syntactic sugar.

As far as we know, this way in which Esp is simultaneously a declarative and a procedural synchronisation mechanism is unique. The only other synchronisation mechanism we know of that offers both declarative and procedural programming styles is Eiffel||. In this, a declarative synchronisation mechanism has been implemented on top of the native, procedural mechanism [Car90b]. However, a limitation of Eiffel|| is that programmers can use *either* the declarative mechanism *or* the procedural mechanism, but not degrade gracefully from one into the other.

4.4 Optimisation Techniques

The numerous examples presented in this chapter show that Esp offers excellent expressive power. It is natural to wonder if the price of this power will be slow execution speed. While this would certainly be true of a naive implementation of Esp, optimisation can improve its execution speed several-fold. Sections 4.4.1 and 4.4.2 outline some approaches that can be taken to optimise Esp code.

4.4.1 Re-evaluation Matrices

Section 4.1.1 on page 47 discussed how re-evaluating guards at every event is sufficient to guarantee the intuitive *as soon as* semantics of guards. However, it is usually unnecessary to re-evaluate guards this often. A guard is expressed in terms of synchronisation variables, which are updated at events. Thus, if a guard evaluates to false then it is necessary to re-evaluate it *only* at those events that update one or more of the synchronisation variables referenced by the guard. Re-evaluating the guard at events other than these would be redundant since the guard could not have changed value.

An optimising compiler could construct a *re-evaluation matrix* that shows at which events guards need be re-evaluated. This matrix could then be used to generate code that avoids unnecessary re-evaluation of guards.

There are two tasks involved in constructing a re-evaluation matrix: (i) determining at what events each synchronisation variable is updated, and (ii) determining what synchronistion variables are used in the evaluation of each guard. We discuss each of these tasks in turn. Then we give an example of a re-evaluation matrix and discuss how it can be optimised to further eliminate unnecessary guard re-evaluations.

4.4.1.1 Determining the Events at which a Synchronistion Variable is Updated

Esp maintains some synchronisation variables automatically, and it is trivial for the compiler

to determine at what events these are updated.

For example, the *arrival*(Foo) synchronisation counter is updated only at the *arrival*(Foo) event. Similarly, other synchronisation counters are updated at their respective events. Recall that the counter *wait*(Foo) is equivalent to the expression "*arrival*(Foo) − *start*(Foo)" so the *wait*(Foo) counter is updated at these two events. Similarly, the *exec*(Foo) counter is updated at the *start*(Foo) and *term*(Foo) events.

Waiting(Foo) and *executing*(Foo) are collections of invocations and thus are updated whenever an invocation is added or removed. Thus, *waiting*(Foo) is updated at the events *arrival*(Foo) and *start*(Foo). Similarly, *executing*(Foo) is updated at the *start*(Foo) and *term*(Foo) events. *Waiting*(Foo) and *executing*(Foo) can also be considered as updated if assignments are made to synchronisation local variables of *Foo* invocations. For example, if the following code were executed in an action then it would be considered to be an update to *waiting*(Foo):

```
for p in waiting(Foo) do
  if p.arr_time < this_inv.arr_time then p.len−−; endif
end
```

It can be more difficult for the compiler to determine at what events programmer-declared synchronisation variables are maintained. Certainly, an assignment statement in an action is an unambiguous indication that a synchronisation variable is being updated at that event. However, it is possible that a variable might be updated, not directly in an action, but rather in a synchronisation operation that is invoked by the action. Thus a compiler might have to perform flow-analysis in order to determine what synchronisation variables are updated at a particular event. If the host language makes it possible for variables to be updated via pointers (and many languages do) then, *in the general case*, flow-analysis might not be able to determine with full accuracy what synchronisation variables are updated at an event. In *more common* cases, however, flow-analysis is likely to be accurate. In those cases where flow-analysis cannot determine what synchronisation variables are updated at a particular event then the compiler should, say, take the pessimistic guess that *all* programmer-declared synchronisation variables are updated at the event in question. In this way, the compiler would not be engaging in any unsafe optimisations.

4.4.1.2 Determining the Synchronisation Variables Accessed in a Guard

The synchronisation variables accessed by a guard can usually be determined by an inspection of the guard. For example, the following guard accesses variables *exec*(Read) and *exec*(Write):

```
Write: exec(Read, Write) = 0;
```

However, if a guard invokes a synchronisation operation then flow analysis must be performed to determine the full set of variables that the guard accesses. As for actions, flow analysis on guards should usually be satisfactory but it is possible that it may prove inconclusive, in

which case the compiler should err on the side of caution when guessing which variables the guard accesses.

4.4.1.3 Optimising the Re-evaluation Matrix

To illustrate the re-evaluation matrix, consider the following guards, taken from the Bounded Buffer example (Figure 4.9 on page 55).

#define num *term*(Put) − *term*(Get)
Put: *exec*(Put) = 0 **and** num < Size;
Get: *exec*(Get) = 0 **and** num > 0;

The guard for *Put* accesses the following synchronisation variables: *exec*(Put), *term*(Put) and *term*(Get). Similarly, the guard for *Get* accesses: *exec*(Get), *term*(Put) and *term*(Get). The re-evaluation matrix obtained from these guards is as follows:

	arrival(Put)	start(Put)	*term*(Put)	*arrival*(Get)	start(Get)	*term*(Get)
Put		√	√			√
Get			√		√	√

Each row in the matrix represents a guard. The columns in the matrix represent events. A tick ("√") in a cell of the matrix indicates that one or more synchronisation variables used by the guard of that row are updated at the event of that column. Thus we note that if an invocation of *Put* is blocked then its guard need be re-evaluated only at the *start*(Put), *term*(Put), and *term*(Get) events.

This re-evaluation matrix is exactly half-full which means that, on average, a blocked invocation will have its guard re-evaluated only half as often as it would have if a re-evaluation matrix had not been constructed. However, this can be improved even further by optimising the re-evaluation matrix itself.

In the solution to the bounded buffer problem, the expression "*exec*(Put) = 0" leads to *Put*'s guard being re-evaluated at the *start*(Put) event. However, this guard evaluation is unnecessary since a *start*(Put) event can only make *exec*(Put) non-zero. Similarly, the guard of *Get* need not be re-evaluated at the *start*(Get) event. Taking this into account, the re-evaluation matrix can be reduced to:

	arrival(Put)	start(Put)	*term*(Put)	*arrival*(Get)	start(Get)	*term*(Get)
Put			√			√
Get			√			√

This optimisation on the re-evaluation matrix has reduced it from being half-full to being just one-third full. Three comments are in order about this optimisation on the matrix.

Firstly, a similar optimisation can be applied to guards that contain the expression "*wait*(*Put*) = 0". In this case, an invocation need not have its guard re-evaluated at the *arrival*(Put) event, since this event can only make *wait*(Put) non-zero.

Secondly, this optimisation can only be applied where the constant 0 is used. For example, if the expression "$exec(\text{Put}) \leq 1$" was given then the guard *would* have to be re-evaluated at the $start(\text{Put})$ event. Nevertheless, the optimisation is worthwhile because the constant 0 is used frequently.

Thirdly, this optimisation works on expressions of the more general form:

$$exec(Op_1) + exec(Op_2) + \cdots + exec(Op_n) = 0$$

In such cases $start(Op_{1..n})$ need not be marked in the re-evaluation matrix.

4.4.2 Optimisation by Transformation

Even allowing for the re-evaluation matrix, it is likely that the evaluation of guards will be expensive relative to the cost of using a less powerful synchronisation mechanism. Therefore, one possible optimisation approach is to recognise that certain patterns of guards/actions implement particular synchronisation policies, and then generate code to implement these policies in a cheaper manner.[2]

For example, consider an object which has three synchronised operations: A, B and C. The following guards specify that A and B execute in mutual exclusion of each other, and that C executes in mutual exclusion of itself:

synchronisation
A: $exec(\text{A, B}) = 0$;
B: $exec(\text{A, B}) = 0$;
C: $exec(\text{C}) = 0$;

If a compiler can recognise the semantics of these guards then it could transform them into appropriate *wait* and *signal* operations on semaphores (one semaphore for A and B, and another for C) surrounding the body of the operations.

Transformations might also be applied to guards with scheduling predicates. For example, an intelligent compiler might recognise that the following guard queues invocations based on the tuple (*len*, *arrival*) and hence generate code to maintain an ordered linked list of invocations:

Print: $exec(\text{Print}) = 0$
and *there_is_no*(p **in** *waiting*(Print): p.len < *this_inv*.len
or p.len = *this_inv*.len **and** p.arr_time < *this_inv*.arr_time);

If implemented, such transformations would offer programmers the high-level expressive power of Esp but with the efficiency of low-level synchronisation mechanisms.

The main obstacle to implementing optimisation by transformation is, of course, having the compiler recognise the semantics of certain patterns of Esp code, especially if the code

[2] Optimisation by transformation has been applied successfully in other contexts: e.g., tuple space operations in Linda [Zen90].

includes actions as well as guards. However, this is not un-surmountable. We will return to this topic in Part III of this thesis to discuss how *generic synchronisation policies* offers a practical approach to this difficult problem.

4.5 Hybrid Variables

In Section 3.4 we said that the variables of an object might be sequential, synchronisation or hybrid, but that, in practice, hybrid variables rarely, if ever, occur. In this section we discuss how hybrid variables, *if* they should occur, can be handled in Esp.

For a variable to be "hybrid" means that it is needed for the sequential functionality of an object and also for its synchronisation. In such cases, we propose that programmers maintain *two* variables in step: one a (sequential) instance variable and the other a synchronisation variable. In effect, we are replacing the hybrid *variable* with a hybrid *concept* that is implemented by two separate variables.

To see how the two variables can be maintained in step, consider an object which contains two operations: A and B. Consider also that the object has a hybrid variable, x, which is to be initialised to zero, incremented every time A is invoked and decremented every time B is invoked.

Code to maintain the two variables in step is shown in Figure 4.13. The sequential variable, x, is maintained inside the bodies of the operations while the synchronisation variable is maintained by similar code at events associated with the corresponding operations.

```
class Foo{
  int   x;
  Foo() { ... x := 0; ... }
  A(...) { ... x ++; ... }
  B(...) { ... x --; ... }
  ⋮
synchronisation
  int   x;
  start(Foo) → x := 0;
  term(A) → x ++;
  term(B) → x --;
  ⋮
}
```

Figure 4.13: Maintainence of sequential and synchronisation variables in step

An important point to note is that the synchronisation variable, x, is maintained *independently* of its sequential counterpart, and vise versa. Thus the synchronisation code does *not* need to access, either directly or indirectly, the (sequential) instance variable, and so we

retain the modularity of having sequential code and synchronisation code segregated.

Earlier, on page 52 of Section 4.2.3, we argued that in the bounded buffer the variable *num* is a synchronisation variable *only* and is *not* required by the sequential code. If the reader does not agree with this and insists that *num* is needed by both the sequential and synchronisation code[3] then it is possible to accommodate this alternative point of view by use of the coding style shown in Figure 4.13: variable x corresponds to *num*, and operations A and B to *Put* and *Get*, respectively.

4.5.1 A Limitation of the Paradigm

We have claimed that the two variables implementing a hybrid concept can be maintained independently of each other. It is, however, possible to think of a *hypothetical* case in which this does not hold true.

The code in Figure 4.13 maintains x using simple and deterministic logic. But consider what would happen if the code to maintain x was quite complex or even non-deterministic. For example, consider the case where the value assigned to x in operation A is dependent on, say, a high-speed clock. Even if the synchronisation code accessed the same clock, the time returned by the clock might differ between successive calls and so there is no guarantee that the synchronisation variable, x, will have the same value as its sequential counterpart.

A work-around in this case is for the sequential code to invoke an operation, say, *DummyOp*, passing x as a parameter, whenever x is updated. The synchronisation code can associate an action with the *start*(DummyOp) event to read the new value of x. This technique is illustrated in Figure 4.14.

In this case the maintenance of the synchronisation variable, x, *is* dependent upon its sequential counterpart. However, we have yet to see a need to employ such techniques and so for the moment it is of hypothetical curiosity rather than of practical concern.

4.6 Summary

This chapter has introduced a synchronisation mechanism, Esp, that embodies the concepts of the Sos paradigm. Through this mechanism, we have illustrated several benefits of Sos, notably:

- Modularity is achieved by completely separating synchronisation code and variables from sequential code and variables.
- The separation of synchronisation code and variables from sequential code and variables elminates the need for synchronisation code to have unsafe access to instance variables.

[3] One possible reason why readers may think that *num* is required by the sequential code is to specify pre- and postconditions [Mey92] on the operations. However, we show in Section 14.5 that it is possible to express pre- and postconditions in the synchronisation part of a class.

```
class Foo {
  int    x;
  A(...)
  { ...
    x := current_time();
    DummyOp(x);
    ...
  }
  DummyOp(int new_x) { }
synchronisation
  int    x;
  start(DummyOp) → x := this_inv.new_x;
  ⋮
}
```

Figure 4.14: Maintaining sequential and synchronisation variables in step under complex or non-deterministic logic

The examples in Section 4.2.3 showed that several synchronisation policies which traditionally have been thought to require access to instance variables can be implemented in terms of synchronisation variables instead.

- The many examples in this chapter show that Esp offers excellent expressive power. In particular, the implementations of starvation-free versions of both the Shortest Job Next scheduler and the Dining Phisohophers problem illustrate how Esp excels in the areas of scheduling and liveness constraints.
- The examples in Section 4.2.1 showed that (i) synchronisation counters, (ii) the relative arrival time of invocations, and (iii) scheduling predicates are forms of syntactic sugar. As such, there are remarkably few constructs essential to Esp. This supports our claim that Sos-based synchronisation mechanisms can achieve expressive power without complexity or creeping featurism.

An interesting aspect of Esp is that it is both procedural *and* declarative, which permits a graceful degradation of expressive power.

Finally, it is natural to assume that the expressive power of Esp might be at the expense of run-time performance. However, this need not be the case as we have outlined some optimisation techniques in Section 4.4 which can reduce the run-time overhead of Esp dramatically.

Chapter 5

Language Support for Esp

The previous two chapters presented the Sos paradigm and Esp (which embodies the concepts of Sos) in a language-independent manner. This style of presentation was used in order to avoid confusing the central concepts of Sos and Esp with the idiosyncrasies of a particular language. This chapter now discusses the issues that arose when we added Esp to the Dee language [Gro90]. There is nothing special about Dee that makes it particularly suitable to act as a host language for Esp.[1] As such, most of the language-design issues discussed in this chapter would be relevant if Esp were to be introduced to another object-oriented language.

One important topic missing from this chapter is how synchronisation code is inherited in Dee. A discussion of this issue is deferred until Part IV which is devoted to the problems associated with the use of inheritance in COOPLs.

Also missing from this chapter is a discussion of how we have implemented Esp on top of Dee. Later, in Part III of this thesis, we show that a language can support not just Esp but also generic synchronisation policies. We defer a discussion of implementation issues until this other topic has been addressed.

5.1 A Brief Overview of Dee

Dee [Gro90] is an object-oriented language developed partly to experiment with the object-oriented paradigm and partly for use as a teaching tool. All data types in Dee are classes, including types such as integers, booleans and strings. Dee supports genericity—arrays, lists, sets and so on are provided as generic classes.

The Dee compiler produces pseudo-code (pcode) for a hypothetical, stack-based machine, and this pcode is then interpreted.

Dee was originally written to run on MSDOS. However, we ported Dee to UNIX and extended the compiler to accept Esp. The combination of Dee and Esp is called "Desp."

[1] The main reason why we chose Dee as a base language for our prototype with was not due to any features of the language itself, but rather because we had access to a Dee compiler that produced pcode. We felt it would be easier to prototype and debug concurrency constructs in an interpreted pcode environment than in a compiled environment. A discussion of the issues surrounding this decision can be found in Section 8.2.

Like Dee, DESP produces pcode.

5.2 Separation of the Synchronisation and Sequential Parts of a Class

As in ESP, the keyword **synchronisation** is used to syntactically separate synchronisation code and variables from sequential code and variables. Semantic checks in the compiler ensure that synchronisation code does not access sequential code or variables, and vice versa.

5.3 Actions and Guards

In merging ESP with Dee, we wanted to make as few changes to the host language as possible so we sought ways in which the core concepts of ESP could be implemented in terms of existing language constructs.

We recognised that actions can be considered to be operations that have the following three unusual characteristics:

(i) They are invoked *automatically* (at events).

(ii) In ESP, actions do not have names. Unfortunately, this prevents programmers from being able to explicitly invoke actions. However, this is a purely syntactic issue and alternative syntax could provide actions with names, thus allowing them to be invoked like other operations.

(iii) Actions implicitly take a parameter—denoted in ESP as *this_inv*—that permits access to information about the invocation for which the action is being executed.

Similarly, guards can be considered to be operations that return a boolean result. Like actions, guards have the following properties: (i) they are invoked automatically by the run-time system; (ii) they do not have names; and (iii) they implicitly take *this_inv* as a parameter. Thus, the issues involved in treating guards as language-level operations are similar to those in treating actions as language-level operations.

Being able to treat actions and guards as normal operations offers the benefit of making it possible to inherit actions and guards. Also, while overloading a syntactic construct with multiple semantics often results in complexity, in this case we feel that merging actions and guards with operations actually simplifies the language.

However, in order to achieve this goal of treating actions and guards as normal operations, it must be possible to treat "invocation" as a language-level type since actions and guards need to explicitly take an invocation as a parameter. We defer discussion on language support for invocations until Section 5.4. For the moment, accept that invocations are represented at the language level by a type called *Invocation*.

The code in Figure 5.1 provides an example of the declaration of an action and a guard in DESP. In this example, the class has three operations (or "methods" in Dee terminology): a

sequential operation called *Bar* and two synchronisation operations called *a_Bar* and *g_Bar*. The **map** directive informs the compiler that the operation *a_Bar* is to be invoked whenever the event *arrival*(Bar) occurs, and similarly operation *g_Bar* is to be invoked as the guard of *Bar*.

```
class Foo

public method Bar(...)
begin ... end

synchronisation
...
private method a_Bar(this_inv: Invocation)
begin ... end

private method g_Bar(this_inv: Invocation): Bool
begin ... end
...
map arrival(Bar) → a_Bar
      guard(Bar) → g_Bar
```

Figure 5.1: Declaration of actions and guards in DESP

The run-time system will create and initialise an *Invocation* object at the *arrival*(Bar) event; this object will then be passed as a parameter to *a_Bar* and *g_Bar*.

A DESP programming convention illustrated in this example is that if a synchronisation operation is to be executed at an *arrival* event then its name is that of the corresponding sequential operation with a prefix of "a_". Similarly prefixes of "s_" and "t_" are used to designate synchronisation operations executed at *start* and *term* events, respectively, and the prefix "g_" is used to designate guards.

5.4 Invocations as Language-level Types

When, in the previous section, we discussed how DESP implements actions and guards in terms of operations, we asked the reader to accept that an *Invocation* language-level type existed. In this section we discuss the issues involved in representing invocations as language-level types.

The main obstacle to having an *Invocation* data type is that there is no *single* type of invocation. Rather, within a class, there are as many different types of invocation as there are operations defined for that class. Since the operations defined within a class may all take different numbers, and types, of parameters, an invocation for one operation is unlikely to

be interchangeable with an invocation for a different operation. This raises the issue of how particular invocation types can be defined.

We discuss several possibilities:

- Represent *Invocation* as a Pascal-style variant record. (This approach is discussed in Section 5.4.1.)
- Represent *Invocation* as a generic class that is instantiated upon operations. (This is discussed in Section 5.4.2.)
- Have an *Invocation* class in which parameters are stored in an untyped list. (This is discussed in Section 5.4.3.)
- Use an inheritance hierarchy to model different invocation types. (This is discussed in Section 5.4.4.)

5.4.1 The Variant-record Approach

In a non object-oriented language, the different *Invocation* types of the operations of a resource could all be combined into a single variant-record type (which would be implicitly declared by the compiler). However, this approach is not without its problems.

One problem is that, in many languages, variant records are not *compile-time* type-safe. (However, if a variant record contains a "tag" field then it is possible to generate code for *run-time* checking to ensure that accessed fields are compatible with a variant record's current "tag".) It *is* possible to guarantee compile-time type-safety by imposing some restrictions so that such variant records could be used only within certain language constructs. Mediators [GC86] and CEiffel [Löh91], both of which represent invocations as variant-records, *appear* to impose such restrictions, though little is said about this in papers on those languages.

Another problem with this approach is that an invocation would be "variant" not only upon the *operation* invoked but also upon the *class* to which the operation belongs. For example, several different classes may have, say, a *Get* operation but the invocation of *Get* for a bounded buffer may not be interchangeable with an invocation of *Get* for an array or a hash table. Thus, an *Invocation* variant-record would require *two* "tag" fields instead of the usual one. A possible approach to keep the number of "tag" fields to one would be to restrict access to an *Invocation* type to within a class; in effect, each class would have its own private *Invocation* type that could not be accessed from outside that class. (CEiffel takes this approach.) However, this would then mean that *Invocation* would not be first-class, i.e., there would be some restrictions on its usage. For example, it might not be possible to pass an *Invocation* as a parameter to an operation of another object.

Another drawback of this approach is that while it might be suited to, say, Pascal or C, it would not suit an object-oriented language since variant records themselves are not object-oriented.[2]

[2] Our argument that variant records are not object-oriented is as follows. If one considers classes to be

Having considered these drawbacks of the variant record approach, we decided against adding variant records to the DESP language as a means of supporting invocation types.

5.4.2 The Generic Class Approach

The Dee language provides support for generic classes. For example, in the following variable declaration, the generic class *List* is instantiated upon the class *Person*.

```
var  employees: List[Person]
```

One might hope that the guard and actions of operation *Foo* could take a parameter of type "Invocation[Foo]". However, to do this would require the ability of a generic class to be instantiated upon *operations*, while the Dee language permits instantiation only upon *classes*.

One approach to overcome this mismatch between the generic class facilities that Dee provides and those needed for this proposal would be to modify the definition of the language to allow a generic class to be instantiated upon *either* operations *or* classes. However, this approach would possibly result in a substantial increase in the complexity of the language and the compiler.

Another approach would be to promote operations to the status of classes in their own right. If this were done then the existing generic class facilities would be sufficient to permit an *Invocation* class to be instantiated upon an operation. While this approach would be feasible for languages that *already* confer class status upon operations, we were unwilling to undertake the task of adding such capabilities to the Dee language, preferring instead to find an alternative way that would not change the language in such a fundamental manner.

5.4.3 The Untyped List Approach

Invocations could be represented by a class such as that shown in Figure 5.2. This approach, which is used in Eiffel|| [Car93], deals with the issue of different operations taking different numbers/types of parameters by storing parameters in a list of *Any* (in DESP all classes conform to the base class *Any* and hence it is an untyped list).

However, such an approach would be inconvenient for programmers since access to parameters would not be by name, but rather by position in the list. Such access would, of course, be error-prone. It also precludes type-checking of parameter access at compile time. Instead, programmers would have to rely on run-time type-checking.

For these reasons we decided against using this approach in DESP.

5.4.4 The Inheritance Approach

The final approach we consider—and the approach we have adopted in DESP—is to use a hierarchy of classes to represent different invocation types.

---

the object-oriented replacement of record types, then class hierarchies are the object-oriented replacement of variant records.

```
class Invocation
inherits Any

public var arr_time: Int      { arrival time of invocation }
public var OpName: String     { name of invoked operation }
...     { any other useful information }
public var Parameters: List[Any]
```

Figure 5.2: Invocation class that stores parameters as a untyped list

A base class, *Invocation*, is shown in Figure 5.3. This class includes some instance variables that programmers may find useful for synchronisation.[3] These instance variables will be initialised by the run-time system when the run-time creates an *Invocation* object (at an *arrival* event).

```
class Invocation
inherits Any

public var arr_time: Int      { arrival time of invocation }
public var OpName: String     { name of invoked operation }
public var ClientId: Int      { ID of invoking process }
```

Figure 5.3: Base *Invocation* class in DESP

The base class, *Invocation*, does not include any parameters because, as we have said earlier, the number and type of parameters can vary from one operation to another. Instead, programmers can subclass from *Invocation* and declare, within the subclass, instance variables that correspond to parameters of an operation. Similarly, if programmers wish to have synchronisation local variables for an operation then these can also be declared as instance variables within a subclass of *Invocation*. We illustrate this subclassing of *Invocation* with an example.

Alarm Clock (Revisited)

Consider the ESP implementation of the Alarm Clock problem shown in Figure 4.5 on page 51 of the previous chapter. The synchronisation code accesses the *period* parameter of operation *WakeUp*. The synchronisation code also declares a variable, *wakeup_time*, local to invocations of *WakeUp*. In translating this ESP code to DESP, we create a subclass of *Invocation* as

[3] Examples in the previous chapter have already illustrated the use of *arr_time*. The other instance variables of the *Invocation* class—*OpName* and *ClientId*—are rarely of use when implementing synchronisation policies *per se*. However, the information they contain can be printed in diagnostic messages at events; this can be useful for debugging and also for pedagogical purposes to illustrate the event-based nature of ESP.

shown in Figure 5.4. This subclass contains *period* and *wakeup_time* as instance variables. The subclass also contains an operation to permit *wakeup_time* to be initialised.

```
class WakeUpInvocation
inherits Invocation

public var period: Int      { copy of a parameter }
public var wakeup_time: Int      { synchronisation local variable }

public method set_wakeup_time(Val: Int)
begin
  wakeup_time := Val
end
```

Figure 5.4: The *WakeUpInvocation* class

With the *WakeUpInvocation* class written, we can now write the code for the *AlarmClock* class. This class, shown in Figure 5.5, is a direct translation from the Esp version (Figure 4.5 on page 51). Note that the guard and *arrival* action of *WakeUp* take a parameter of type *WakeUpInvocation*. The compiler will note that the *period* parameter of *WakeUp* has a namesake in an instance variable of *WakeUpInvocation* and will generate code to copy this parameter at run-time.

5.4.5 Discussion

Unlike the variant record approach (Section 5.4.1) and the technique of accessing parameters by their position in an untyped list (Section 5.4.3), using inheritance to model different invocation types provides type-safe access to parameters. This approach also has the advantage of not requiring extensive modifications to the host language, Dee, unlike the generic class approach (Section 5.4.2).

However, the inheritance approach has its own disadvantage. For synchronisation code to be able to access a parameter of an operation requires that a namesake of that parameter be declared as an instance variable in a subclass of *Invocation*. In effect, the parameter is declared twice: once in the signature of the operation and again as an instance variable in an *Invocation* subclass. Such duplicate declarations are undesirable because of the possibility that, through accidental error, the two declarations might not be identical. In particular, if the name of the instance variable in the *Invocation* subclass is misspelt then the compiler will assume that it is intended to be a synchronisation local variable rather than a copy of the parameter. In such cases a program might compile but give a run-time error due to access of the uninitialised instance variable of the *Invocation* subclass.

Luckily, programmers have *some* protection against this danger. For the compiler to fail to detect a misspelling in the declaration of the instance variable of an *Invocation* subclass

```
class AlarmClock
inherits Any

public method WakeUp(period: Int)      begin end
public method Tick      begin end
synchronisation

private method a_WakeUp(t: WakeUpInvocation)
begin
  t.set_wakeup_time( t.period + term(Tick) )
end

private method g_WakeUp(t: WakeUpInvocation): Bool
begin
  result := term(Tick) >= t.wakeup_time
end

map arrival(WakeUp) → a_WakeUp
      guard(WakeUp) → g_WakeUp
```

Figure 5.5: DESP implementation of an Alarm Clock

but still successfully complete compilation would require that *all* accesses to this instance variable be similarly misspelt. Such consistent misspelling is somewhat unlikely and so the chances are that the compiler would report an error.

The problem could be corrected by introducing a language construct which would inform the compiler that a particular instance variable of an *Invocation* subclass is to be a copy of a parameter of an operation. However, this seems to be an inelegant approach. Later, in Part III, we discuss how language support for generic synchronisation policies can solve this problem in a more natural manner.

The reader may be concerned that concurrent programs will contain an overwhelming number of *Invocation* subclasses. While we do not yet have enough experience of writing DESP programs to know for sure, we feel that this concern is misplaced for several reasons.

Firstly, it is likely that the majority of classes in a concurrent program will be sequential; only a few classes will contain synchronisation code.

Secondly, of the few classes in a program that are synchronised, most are likely to implement synchronisation policies that do not require synchronisation local variables (other than *arr_time* which is maintained automatically) or access to parameters of invocations. For these synchronised classes, the base *Invocation* class will suffice.

This leaves only a small fraction of classes within a program that need to utilise subclasses

of *Invocation* to implement their synchronisation policies. Even among these classes, there may be potential for reusing subclasses of *Invocation*.

For instance, the constructor operation of a bounded buffer might take a *size* parameter that determines the capacity of the buffer. The synchronisation code of the buffer is likely to need to access this *size* parameter and so the programmer will write a *SizeInvocation* class (as shown in Figure 5.6) for this purpose. Once *SizeInvocation* has been written it can be reused by other synchronised classes whose constructors also take a *size* parameter. (Such classes might include hash and symbol tables, disk head schedulers and a dining table whose capacity is not, as tradition has it, fixed at five philosophers.)

```
class SizeInvocation
inherits Invocation

public var size: Int      { copy of a parameter }
```

Figure 5.6: The *SizeInvocation* class

5.5 Safe Access to Parameters

The SOS paradigm requires that synchronisation code have a safe way to access parameters of invocations. One way mentioned in Section 3.5.1 was for the run-time system to arrange for the parameters of an invocation to be copied when the invocation arrives. These parameter *copies* could be accessed by the synchronisation code, safe in the knowledge that if the sequential body of an operation updated a parameter then this would not affect the synchronisation code's copy of that parameter.

We would have liked to have used this technique in DESP. Unfortunately, a "copy" operation is not provided as standard in Dee classes, thus making it difficult for the run-time system to copy parameters. Instead, the approach we have taken is for synchronisation code and sequential code to share parameters. Ensuring that this sharing is safe has necessitated the following restriction: synchronisation code can access parameters only of types *Int*, *Bool*, *String* and *Float*. A look back at the numerous ESP examples presented in Section 4.2 will show that the only parameters accessed by synchronisation code happen to be one of these four basic types.[4] As such, it seems that this restriction will not be a hindrance in practice.

There are two unusual aspects of the classes *Int*, *Bool*, *String* and *Float* that make it safe for parameters of these types to be shared.

The first is that these classes do not have any mutator operations, i.e., operations that will change the state of the object. Rather, operations that apparently change the value of,

[4]The only exception to this is the Dynamic Priority Print Queue (Figure 4.8 on page 54) in which synchronisation code accesses a parameter of type *GroupId* which is an enumerated type. However, enumerated types are, in effect, integers.

say, an *Int* object actually create a new *Int* object that contains the desired value and returns that instead [Gro90]. (The old object will, if necessary, be garbage collected.)

The second unusual aspect of these classes is the restriction that subclasses of them cannot be written. (This is a restriction that exists in Dee rather than being a restriction that we introduced.)

The combination of these two unusual aspects of the classes *Int*, *Bool*, *String* and *Float* guarantee that parameters of these types cannot be mutated and hence can be safely shared by sequential code and synchronisation code.

5.6 New Language Constructs

The *waiting* and *executing* constructs of ESP were added to the language as built-in functions which return "Collection[(subtype of) Invocation]". "Collection" is a generic type, in Dee's standard library, which can be iterated over. As the existing Dee loop construct was somewhat awkward to use, some new iterator loops were added as syntactic sugar, including the **for** loop statement and the *there_is_no* predicate as illustrated in the ESP examples of Section 4.2.

The syntax of these DESP loop constructs is similar to their ESP equivalents. To illustrate this, you can compare the DESP implementation of the Starvation-free, Shortest Job Next scheduler in Figure 5.7 with the ESP version shown previously in Figure 4.6 on page 52. Aside from the more verbose syntax, some small differences to note are as follows.

Firstly, in ESP the **for** loop and scheduling predicates implicitly declared their own loop variable. In DESP this variable must be declared explicitly.

Secondly, *this_inv* is, in effect, a keyword in ESP. However, since this is passed as a parameter to guards and actions in DESP, it loses its reserved-word status and becomes a "normal" parameter name. As such, programmers can refer to it by whatever name they want. For example, in Figure 5.7 it is referred to by the shorter name *t*.

Finally, since our presentation of ESP in Chapter 4 was language-independent, it did not address the issue of how invocations might be represented at the language level. In hindsight it should be clear to readers that the ESP pseudo-code representation of invocations was akin to the variant-records approach discussed in Section 5.4.1. As already discussed in Section 5.4.4, the approach adopted in DESP is to have a hierarchy of classes with *Invocation* as its root. Thus the DESP implementation of synchronisation policies that require either access to parameters or the ability to maintain synchronisation local variables will necessitate the writing of an appropriate subclass of *Invocation*. This is illustrated in the implementation of the Starvation-free, Shortest Job Next scheduler in Figure 5.7, which makes use of the *LenInv* class shown in Figure 5.8. As well as defining an instance variable, *len*, to provide access to its namesake parameter, the *LenInv* class also defines an operation, *set_len* to set the value of this instance variable. This operation is called by the *start*(Print) action (operation *s_Print*) to decrement the *len* of invocations that have been skipped over.

```
class FairSJN   inherits Any
cons make(...)   { constructor }
begin ... end
method Print(fileName:String   len: Int)
begin ... end
synchronisation
method s_Print(t: LenInv)
var   p: LenInv
begin
  for p in waiting(Print) do
    if p.arr_time < t.arr_time then p.set_len(p.len − 1); endif;
  end;
end

method g_Print(t: LenInv): Bool
var   p: LenInv
begin
  result := (exec(Print) = 0) and (there_is_no(p in waiting(Print):
                                  p.len < t.len or p.len = t.len and p.arr_time < t.arr_time));
end

map start(Print) → s_Print
      guard(Bar) → g_Print
```

Figure 5.7: Starvation-free, Shortest Job Next scheduler in DESP

```
class LenInv
inherits Invocation

public var len: Int      { copy of a parameter }

public method set_len(Val: Int)
begin
  len := Val
end
```

Figure 5.8: The *LenInv* class (used in the starvation-free SJN scheduler).

5.7 "Super" Calls are Unsynchronised

Consider a base class that defines an operation, *Foo*, and a guard for that operation. A subclass will inherit both operation *Foo* and its guard, and can re-implement these independently of one another. For example, if a subclass re-implements operation *Foo* then the guard will apply to this re-implemented version of *Foo* in the subclass rather than the base class version of *Foo*. A consequence of this is that any "super" calls to the version of *Foo* in the base class will not be synchronised.

The details of this may seem confusing at first, but the end result is intuitive semantics for "super" calls. For example, consider a subclass that inherits the guard "*exec*(Foo) = 0" for operation *Foo*, and re-implements *Foo* in the following manner:

```
method Foo(...)
begin
  super.Foo(...);[5]
  ...      // some new code
end
```

If the guard had applied to both the subclass version and the base class version of *Foo* then the "super" call would have resulted in deadlock due to the nested attempt to gain exclusive access to *Foo*. As such, it is vital that the "super" call be unsynchronised.

Note that language rules preclude the possibility of a client making a "super" call upon a service object; thus there is no danger of a client using "super" to bypass the synchronisation constraints of a service object.

5.8 Implementation and Run-time Overhead of Synchronisation

Later, in Part III of this thesis, we extend DESP to provide support for generic synchronisation policies. There is some overlap between the implementation issues of concern to ESP and those of concern to generic synchronisation policies. As such, we defer discussion of implementation issues until Part III when we can examine all such issues together rather than have such a discussion split in two parts with some repetition.

We also defer until Part III an analysis of the run-time overhead of synchronisation policies implemented in ESP. This is for two reasons. Firstly, this analysis depends on the discussion of our prototype implementation. Secondly, generic synchronisation policies make possible an interesting optimisation technique that greatly reduces the run-time overhead of synchronisation.

[5] In DESP, "super" calls are made by prefixing the operation to be invoked not with "super" but with the name of the parent class. However, in this thesis we denote such calls with "super" since this is more likely to be familiar to readers.

In the meantime, all we will say about the run-time overhead of synchronisation policies implemented in Esp is that, for an unoptimised implementation of Esp, this overhead can be in the order of several hundred CPU instructions, i.e., it can take several hundred instructions more to invoke a synchronised operation than it takes to invoke an unsynchronised operation. This overhead makes (unoptimised implementations of) Esp unsuitable for use in applications that utilise fine-grained concurrency because this overhead of synchronisation would counteract the potential speed-up of concurrent execution. However, Esp may be useful in applications that utilise coarse-grained concurrency, i.e., applications in which processes do a lot of independent computation and synchronise infrequently.

Of course, optimisation techniques that significantly reduce the overhead of synchronisation may make Esp suitable for use in applications that utilise medium- and fine-grained concurrency.

5.9 Summary

The Sos paradigm consists of four concepts:

(i) Actions executed at events.

(ii) A means of causing an invocation to *start* executing. Esp employs guards for this purpose.

(iii) Access to information about invocations.

(iv) A separation between the sequential code/variables of a class and its synchronisation code/variables.

These concepts have been incorporated into the Dee language in the following manner. The introduction of a single new construct, **map**, enables both (i) actions and (ii) guards to be expressed as operations. Support for (iii) access to information about invocations is provided via the combination of a library class, *Invocation*, and the new language construct of *waiting*/*executing* expressions. Finally, a new keyword, **synchronisation**, enforces (iv) the separation of sequential code/variables from synchronisation code/variables.

Thus, surprisingly few changes were required in order to introduce the Sos paradigm to the Dee language.[6] Moreover, the Desp language is backwards compatible with Dee source code. This is important since it means that Desp programmers can utilise existing libraries of Dee classes.

By introducing only a few changes to the host language and also maintaining backwards compatibility with existing source code libraries, we feel we have shown that the concepts of the Sos paradigm can be applied relatively easily to an object-oriented language.

[6] We did, of course, also introduce synchronisation counters and scheduling predicates but, as shown in Section 4.2.1, these are simply forms of syntactic sugar, rather than being a core part of the Sos paradigm or Esp. Even counting these forms of syntactic sugar, the changes introduced to the host language were few.

Chapter 6

Related Work and Summary of Contributions

In Part II of this thesis we have presented the SOS paradigm in a top-down fashion. We started with a discussion of the core concepts of the paradigm; then introduced a sample synchronisation mechanism, ESP, that embodied these concepts; finally we added ESP to a host language in order to show that the concepts of the SOS paradigm can be introduced to a language in a straight-forward manner.

The bulk of this chapter, which draws Part II of the thesis to a close, is devoted to a discussion of related work. We start off in Section 6.1 by comparing the SOS paradigm with synchronisation paradigms proposed by other researchers. Aside from ESP, we know of just one other synchronisation mechanism that (almost) fully embodies the concepts of the SOS paradigm. We review this mechanism, Mediators, in Section 6.2. Then in Section 6.3 we discuss how support for the concepts of SOS can be found, albeit piecemeal, in a variety of other synchronisation mechanisms. This chapter closes with a summary of the contributions of the SOS paradigm and a discussion of some areas for future work, in Sections 6.4 and Section 6.5, respectively.

6.1 Other Paradigms for Synchronisation Mechanisms

Most papers about synchronisation in COOPLs concern themselves with specific synchronisation mechanisms. The literature is quite sparse on synchronisation *paradigms*—especially ones that are comparable to SOS. However, we did come across two such paradigms; we discuss them in Sections 6.1.1 and 6.1.2.

6.1.1 Synchronising Actions

The *Synchronising Actions* (SA) paradigm [Neu91] contains concepts that correspond approximately to the core concepts of SOS. Thus, it makes sense to compare these two paradigms on the basis of these concepts.

The first concept of the SOS paradigm is that of events and the code (actions) executed at them. SA is also event-based. Code executed at a *start* event is, in the parlance of SA, a *pre_action* and code executed at a *term* event is a *post_action*. However, SA does not recognise the *arrival* event and, consequently, does not permit programmers to associate an action with it. This limits the expressive power of SA. For example, in Section 4.2.2 we showed ESP implementations of some synchronisation policies—starvation-free versions of both the Shortest Job Next scheduler and the Dining Philosophers problem, and the Alarm Clock problem—that utilised *arrival* actions. These cannot be implemented easily in SA.

The second concept of SOS is the requirement that a synchronisation mechanism must provide a way to cause an invocation to start executing. SA has a similar requirement.

The third core concept of the SOS paradigm is that synchronisation code should have access to information about invocations. SOS actually makes five requirements (discussed in Sections 3.3.2 and 3.5) on the access that synchronisation code has to parameters. These requirements are made in order to ensure modularity, safety and expressive power. The SA paradigm also grants synchronisation code access to invocations. However, it does not place any requirements on the type of such access.

The final concept of SOS is that synchronisation code and variables should be separated from sequential code and variables. There are two reasons for this requirement. Firstly, it is *unsafe* for synchronisation code to access the instance variables of an object. Secondly, it is also *unnecessary* since synchronisation code can maintain whatever variables it needs itself without sacrificing expressive power.

Like SOS, SA requires that synchronisation code and variables be separated from sequential code and variables. Neusius (the designer of the SA paradigm) is aware that it is unsafe for synchronisation code to access instance variables.[1] However, there is nothing in the presentation of the SA paradigm to indicate an awareness that such access is also unnecessary. As such it is not clear that Neusius fully appreciates the nature of synchronisation variables. In particular, although SA embraces the concept of synchronisation variables of an object, there is no explicit support for the other types of synchronisation variables: parameters and local variables.

In summary, the SA paradigm contains the same concepts as the SOS paradigm. However, it does not embrace them fully and thus is not as powerful.

In drawing this comparison to a close, we note that the SA paradigm contains two restrictions. The first restriction is a synchronised object must dedicate a thread of control to executing its synchronisation code. The second is that invocations are made via message-passing. Neither of these restrictions are in SOS.

[1] The awareness that it is unsafe for synchronisation code to access instance variables is indicated by the following comment about guards in the Guide language: "[A guard] uses instance variables that represent the internal state of the object. A programmer has to consider the consistent change of these variables within the [body of an operation]" [Neu91, pg. 122].

6.1.2 Thomas's Generic Model

Thomas [Tho92] proposes a paradigm for synchronisation mechanisms. Unfortunately, he neglects to name it; instead he just refers to it as a "generic model."

As with SA, we use the core concepts of the Sos paradigm to compare Thomas's generic model (Tgm) with Sos.

The first concept of Sos is that of actions executed at events. Tgm recognises the existence of the three events—*arrival*, *start* and *term*. However, it makes the restriction that a synchronisation mechanism can associate an action with only one of them. No justification is offered for this restriction which limits expressive power.

The second concept of Sos is that a synchronisation mechanism must have a construct to cause an invocation to start executing. Sos does not make any restriction about what form such a construct might take. Tgm is more restrictive, requiring that guards be employed for this purpose.

The third concept of Sos is the requirement that synchronisation code have access to information about invocations. Tgm does not have a similar requirement.

The final concept of Sos is the requirement that synchronisation code and variables be separated from sequential code and variables. Tgm does not make any such requirement. Indeed, it explicitly states that synchronisation code *should* have access to the instance variables of an object. This requirement of Tgm is made in the mistaken belief that access to instance variables is required in order to implement some synchronisation policies. This access is unsafe since Tgm does not take any steps to ensure that the synchronisation code accesses instance variables only when they are in a consistent state.

In summary, it is clear that Tgm is extremely limited in comparison to Sos. Its poor support for events coupled with its lack of support for access to information about invocations results in limited expressive power, and its access to instance variables is unsafe. In closing, we note that Tgm has a restriction in common with SA: it specifies that a synchronised object must dedicate a thread of control to executing its synchronisation code.

6.2 Mediators: A Synchronisation Mechanism that Embodies (Most of) the Sos Paradigm

In this thesis we have presented a sample synchronisation mechanism, Esp, that embodies all the concepts of the Sos paradigm. We are not aware of any other existing synchronisation mechanism that also embodies all the concepts of Sos. However, one mechanism, Mediators [GC86], comes quite close in doing so—much closer than any other mechanism of which we are aware. In this section we give an overview of the Mediators mechanism and then discuss how it relates to Sos and Esp.

6.2.1 An overview of Mediators

The Mediators mechanism is quite complex. In order to communicate the important ideas of Mediators without its complexity, this section presents a stripped-down version of Mediators. The syntax we use here is based on that used throughout this thesis and in some places is quite different to that employed in Mediators. For readers who are concerned about such issues, these differences in syntax can be found in Section 6.2.2, along with a discussion of some of the more complex features of Mediators.

Mediators separate synchronisation code and variables from sequential code and variables. This separation is indicated by the keyword **mediator** in a manner similar to the way we have used the keyword **synchronisation**.

A mediated (synchronised) object has its own thread dedicated to executing its synchronisation code. This thread causes an invocation to start executing via one of the statements: **spawn** or **exec**. The **exec** statement causes the mediator thread to service the invocation itself, while the **spawn** statement forks off a thread to service the invocation asynchronously. When an invocation has been serviced, a **release** statement must be executed. This returns results to the client and removes the invocation from the mediator.

Mediators is built around the concept of guarded commands [Hoa78, Dij75]. In brief, a guard can be associated with a list of commands, i.e., statements. The commands will be executed whenever the guard becomes true. The notation used to indicate this is as follows:

guard $\rightarrow$ commands

This notation is similar to that used in SOS to indicate that an action should be executed at an event. Indeed, Mediators treats *arrival* and *term* as conditions rather than events. (As we discuss later, there is no concept of a *start* condition in Mediators.)

The *arrival* condition is true if there is a pending invocation that has not been *fired*, i.e., not had a guarded command executed on its behalf. This concept of a condition being *fired* may sound strange but, in effect, it means that the *arrival* condition only happens once for a particular invocation. This is akin to there being only one *arrival* event for an invocation in the SOS paradigm. Similarly, the *term* condition is true only if there is a terminated invocation that has not been fired.

The *arrival* and *term* conditions implicitly take a parameter akin to *this_inv* in ESP. Since *arrival* and *term* are conditions, they may be combined with other conditions. It is these combined conditions that are most often used as the guards in guarded commands.

The commands of a guarded command are executed in mutual exclusion. (This is akin to SOS where actions are executed in mutual exclusion.)

We give some examples to illustrate the Mediators concepts we have already discussed and to introduce some additional concepts.

Basic Readers/Writer Policy

The code in Figure 6.1 shows the basic readers/writer policy implemented in Mediators. The mediator employs an **init** clause to initialise a variable, *NumReaders*, which is maintained explicitly since Mediators does not automatically maintain synchronisation counters.

```
class ReadersWriter {
  Read(...) { ... }
  Write { ... }
mediator
  int  NumReaders;
  init NumReaders := 0; end_init

  arrival(Write) and NumReaders = 0 → { exec this_inv; release this_inv; }

  arrival(Read) → { NumReaders ++; spawn this_inv; }

  term(Read) → { NumReaders --; release this_inv; }
}
```

Figure 6.1: Mediator implementation of the readers/writer policy

Recall that the **exec** statement, which is used to service invocations of *Write*, causes the mediator thread to service the invocation itself. Hence, whenever a *Write* invocation is being serviced, the mediator is, in effect, blocked and there is no possibility of any other guards being evaluated or their commands being executed. Thus, in terms of synchronisation counters, the constraint "*exec*(Write) = 0" is implicit in the guard of every guarded command. This means that the effective guard for *Write* is:

arrival(Write) **and** NumReaders = 0 **and** *exec*(Write) = 0

Similarly, the effective guards for *Read* are as follows:

arrival(Read) **and** *exec*(Write) = 0
term(Read) **and** *exec*(Write) = 0

This is clearly an example of *arrival* being used in a manner analogous to an Esp-like guard. When such a guard becomes true then an Esp-like *start* event is implicit and any code immediately preceding an **exec** or **spawn** statement is analogous to an Esp-like *start* action.

Note that because invocations of *Read* are serviced asynchronously, via the **spawn** statement, a *term*(Read) guarded command is required in order to perform post-processing of such invocations. A *term*(Write) guarded command is not required since *Write* invocations are serviced synchronously; hence any necessary post-processing statements are written directly following the **exec** statement.

Shortest Job Next scheduler

Mediators does not automatically maintain lists of invocations over which synchronisation code can iterate. Instead, programmers have to maintain such data-structures themselves. This is illustrated by the code in Figure 6.2 which implements a Shortest Job Next scheduler.

```
class SJN {
  Print(int len, ...) { ... }
mediator
  boolean  busy;
  Queue  queue;
  init busy := false; queue := empty_queue; end_init

  arrival(Print) → { queue.insert(this_inv, this_inv.len); }

  term(Print) → { busy := false; release this_inv; }

  not queue.empty() and not busy →
  { foo := queue.remove_head(); busy := true; spawn foo; }
}
```

Figure 6.2: Mediator implementation of the Shortest Job Next scheduler

Whenever an invocation arrives, it is inserted into a queue that is maintained by the programmer. Items are removed from the queue and serviced whenever the following guard is true:

not queue.empty() **and not** busy

The boolean variable *busy* denotes whether or not an invocation is currently executing the *Print* operation.

This example demonstrates that any condition can serve as a guard, not just those that contain *arrival* or *term*. This example also illustrates that a guard consisting of just *arrival* (i.e., not combined with any other conditions) is analogous to an ESP-like *arrival* event.

Alarm Clock

The readers/writer example illustrated how *arrival* can be combined with other conditions and used in a manner analogous to an ESP-like guard and *start* event. Then the SJN example showed how it can be used in a manner analogous to an ESP-like *arrival* event. However, the *arrival* condition cannot be used for both these purposes at the same time. This restriction results in an unusual implementation of the Alarm Clock problem, as shown in Figure 6.3.

A synchronisation variable, *now*, is used to record the passage of time. This is initialised to zero and incremented for every invocation of *Tick*. Each invocation of *WakeUp* has its own

```
class AlarmClock {
  WakeUp(int period) { }
  Tick { }
mediator
  int   wakeup_time local to WakeUp;
  int   now;
  init now := 0; end_init

  arrival(WakeUp) → { this_inv.wakeup_time:= now + this_inv.period; spawn this_inv;}

  term(WakeUp) and now >= this_inv.wakeup_time → { release this_inv; }

  arrival(Tick) → { now ++; exec this_inv; release this_inv; }
}
```

Figure 6.3: Mediator implementation of the Alarm Clock policy

synchronisation local variable,[2] *wakeup_time*. The value of this is calculated when a *WakeUp* invocation arrives. Having used the *arrival*(WakeUp) condition to achieve this, the mediator cannot delay the start of execution of the invocation (unless it maintained a queue of pending *WakeUp* invocations and added it to that) so it starts it immediately and instead delays its *termination* by combining the following constraint onto the *term*(WakeUp) guard:

now >= *this_inv*.wakeup_time

6.2.2 Further Details of Mediators

The discussion in Section 6.2.1 presented a simplified version of Mediators in order to communicate its important ideas without its complexity. This section briefly mentions some of the other features of Mediators, and indicates important differences between the pseudo-code notation we have used in discussing Mediators and the actual Mediators syntax. Readers not interested in Mediators can skip this section as it is intended primarily as an aid to those who may be considering reading the original Mediators paper [GC86].

The Mediators equivalent of *this_inv* is denoted by a variable of type *client_process_id*. As the type name suggests, this is actually a reference to a process rather than an invocation. However, in a mediated object there is a one-to-one mapping between client processes and client invocations; hence a reference to a client process is synonymous with a reference to the current invocation by that client. Programming examples in the Mediators paper declare variables of type *client_process_id* with names such as *i* or *j*. Access to information about an

[2] The Mediators paper [GC86] states that synchronisation local variables are supported. Unfortunately, it neglects to indicate what syntax should be used to declare them. We declare them using the same syntax as used in ESP.

invocation is by means of the construct *job(i)*. For example, a parameter, *length*, is accessed as *job(i).length* while the *service* field indicates the name of the invoked operation.

The syntax of *arrival* (*req* in Mediators terminology) and *term* conditions is also different to what we have indicated in Section 6.2.1. These conditions are parameterised not by the name of an operation but rather by a a *client_process_id* variable.

The following example illustrates these syntactic differences. The guarded command below is written in the pseudo-code notation used in the examples in Section 6.2.1:

$$arrival(\text{Foo}) \rightarrow stmt\ list$$

The real Mediators syntax for the above is as follows:

$$req(\text{i});\ \text{job(i)}.service = \text{Foo} \rightarrow stmt\ list$$

Note that the semicolon after "*req*(i)" should be read as a boolean **and** operator.

In Mediators, guarded commands are not written free-style as shown in the examples in Section 6.2.1, but rather are enclosed in a **cycle** construct which takes the form:

cycle
  $\text{guard}_1 \rightarrow \text{stmt list}_1$
  □
  $\text{guard}_2 \rightarrow \text{stmt list}_2$
  □
  ...
  □
  $\text{guard}_n \rightarrow \text{stmt list}_n$
until *condition*;

This cycles (i.e., loops) continuously, executing guarded commands until its *condition* evaluates to **true**. A translation of the examples of Section 6.2.1 into Mediators syntax would place the guarded commands inside a **cycle** construct with a **false** condition.

In Section 6.2.1 we said that a mediated object has its own thread dedicated to executing its synchronisation code. Actually, a mediated object can have *several* threads executing its synchronisation code. The code for each thread consists of a **cycle** construct and they are separated by "//". However:

> "Only one thread of control is active at a time. The active control block can change only when guards are evaluated. This creates mutually exclusive execution of the statement lists between guard evaluations" [GC86, pg. 473].

Grass and Campbell go on to state that the introduction of multiple threads does not introduce any extra power. Rather multiple threads are available in the belief that they make it easier to implement some synchronisation policies:

> "It is possible to rewrite [multi-threaded mediator code] as one large [single-threaded **cycle** construct]. The resulting [code] is considerably more bulky and actually less clear" [GC86, pg. 474].

We disagree with this assessment. The readers/writer, Shortest Job Next and the Alarm Clock examples in Section 6.2.1 are all implemented using a single thread. If these implementations are expressed in real Mediators syntax then none of them are longer (in lines of code) than the multi-threaded versions of the examples in the Mediators paper [GC86]. In fact, two of them are shorter. We also happen to find the single-threaded versions easier to understand but this could be a subjective preference for one programming style over another.

6.2.3 Discussion

Having presented an overview of the important concepts of Mediators, we now discuss how it relates to the Sos paradigm and the Esp synchronisation mechanism.

One of the core concepts of Sos is the requirement that a synchronisation mechanism has a way to cause an invocation to start executing. Mediators provides two: the **exec** and **spawn** statements.

A second core concept is the requirement that synchronisation code and variables be separated from sequential code and variables. Mediators satisfies this requirement and in doing so provides programmers with the ability to declare synchronisation variables of the object and also synchronisation variables that are local to invocations.

A third core concept is that a synchronisation mechanism should have access to information about invocations. Sos actually makes five requirements regarding such access. These requirements, and how Mediators addresses them, are as follows:

1. All the information about an invocation, that a synchronisation mechanism has access to, should be grouped together as a unit.

   The expression "job(i).foo" can be used to access a parameter or a synchronisation local variable, *foo*, from the invocation denoted as *job(i)*. Similarly, "job(i).service" can be used to determine which *service* (operation) is being invoked. Thus, it is clear that *job(i)* groups together information about an invocation.

2. An action should have access to information about the invocation for which it is being executed.

   Mediators does not support events and actions *per se*, but rather is based on the concept of guarded commands which are similar in some ways. These guarded commands have access to such information.

3. Programmers should be able to declare synchronisation variables local to invocations.

   Mediators provides this ability.

4. It is dangerous for synchronisation code to access the parameters of an invocation while they were being updated by sequential code. A synchronisation mechanism must provide some means to prevent such dangerous access.

   The paper on Mediators does not *explicitly* address this issue. However, it is possible that an implementation of Mediators might address it satisfactily as a side-effect of the implementation approach. For example, the run-time system might provide synchronisation code with a *copy* of parameters. Alternatively, the host language might make a restriction that the only type of parameters that can be accessed by both sequential code and synchronisation code are those that are declared to be read-only (akin to "in" parameters in Ada).

5. Information about invocations should be maintained in a list/collection over which synchronisation code can iterate in order to compare (and, if need be, update) information about invocations.

   Mediators provides this capability. As was illustrated by the Mediators implementation of the Shortest Job Next scheduler in code of Figure 6.2, programmers are responsible for maintaining lists of invocations themselves. This is different from Esp in which the run-time system automatically maintains lists of *waiting* and *executing* invocations. The difference between the two approaches is one of ease of use. Esp performs more "housekeeping" work in this regard, thus relieving programmers of the burden.

The final core concept of the Sos paradigm is that of event-based programming. Mediators does not support this, *per se*. However, it does support guarded commands which are similar in some respects, as we now discuss.

Mediators' *term* condition can be used in a manner analogous to the *term* event of Esp. If Mediators' *arrival* condition is the sole condition in a guarded command then it is analogous to the *arrival* event in Esp. Alternatively, if Mediators' *arrival* is combined with other conditions then it can be used in a manner that is analogous to the *start* event in Esp. Thus, the three events of Esp (*arrival*, *start* and *term*) can be simulated via guarded commands—but not all at the same time since Mediators' *arrival* condition is overloaded to provide *either* the *arrival* event *or* the *start* event. This limitation effectively cripples Mediators support for event-based programming.

Since Mediators' guarded commands cannot effectively *emulate* event-based programming, the important issue then becomes whether guarded commands offer an effective *alternative* instead. After all, one might reason, while Mediators is deficient in its support for event-based programming, it does offer some features not found in Sos. Perhaps these facilities unique to Mediators gives it some expressive power that Sos lacks. Examples of two features unique to Mediators are as follows:

- Sos permits actions to execute only at events. Mediators permits guarded commands to execute at *any* condition, not just those that denote events. This is illustrated in the

Mediators implementation of the Shortest Job Next scheduler (Figure 6.2 on page 85) where code is executed at the condition "**not** queue.empty() **and not** busy".

- The Esp mechanism employs guards to trigger the transition from *arrival* to *start* events. This is the only situation where guards can be used. Guards cannot, for example, be associated with a *term* event. Mediators, on the other hand, does permit this, as is illustrated in the implementation of the Alarm Clock problem (Figure 6.3 on page 86).

However, in both these cases the features unique to Mediators are being used to compensate for the lack of proper support for event-based programming. There is nothing to suggest that these features offer any power, above that of Sos, useful for the purpose of synchronisation.

6.2.3.1 Comparison of Esp and Mediators

Both Esp and Mediators offer excellent expressive power. However, Esp offers two advantages over Mediators.

The first is simplicity. The Sos paradigm, which forms the basis for Esp, is composed of just four concepts. Mediators, on the other hand, contains a multitude. Like Sos, it (i) separates synchronisation code and variables from sequential code and variables. Sos requires that a construct be provided to cause invocations to start executing; Mediators provides both (ii) **spawn** and (iii) **exec** for this purpose. (These are significantly different to one another since the latter implicitly adds a constraint to the conditions of all guarded commands.) Like Sos, Mediators provides (iv) access to information about invocations. Mediators provides guarded commands in which two distinct types of conditions can be used: (v) "normal conditions" and (vi) those that can be *fired* just once, i.e., *arrival* and *term*. Finally, Mediators offers (vii) multi-threading (as briefly mentioned in Section 6.2.2), even though it does not offer any extra power over single-threaded Mediator code [GC86, pg. 474]. Thus, Mediators contains almost twice as many concepts as Sos.

The second area in which Esp has an advantage over Mediators is clarity and conciseness of code. The Mediators implementations of policies such as basic Readers/Writer, Dining Philosophers, Alarm Clock and Shortest Job Next scheduler [GC86] are typically *seven* times longer (in terms of lines of code) and considerably less intuitive than the equivalent Esp implementations.

6.3 Support for Individual Concepts of the Sos Paradigm in Other Synchronisation Mechanisms

As Section 6.2 showed, the Mediators synchronisation mechanism embodies almost all of the Sos paradigm. Aside from Esp, which is fully compliant with Sos, we are not aware of any other synchronisation mechanisms that comes so close to embracing all the concepts of Sos.

However, the concepts of Sos *can* be found piecemeal in other synchronisation mechanisms, albeit sometimes implicitly, as we discuss in Sections 6.3.1 to 6.3.4.

6.3.1 Causing Invocations to Start Executing

One of the four concepts of the Sos paradigm is that a synchronisation mechanism has a way to cause an invocation to start executing. *Every* synchronisation mechanism employs a mechanism to do this and thus every synchronisation mechanism contains at least this one concept of the Sos paradigm.

6.3.2 Events and Actions

The Sos paradigm is event-based and specifies that code, referred to as *actions*, can be executed at events.

The most common form in which events appear in synchronisation mechanisms is probably in the guise of synchronisation counters. These are simply counts of how often particular events have occurred and they are maintained by trivial actions executed automatically by the run-time system. However, most counter-based mechanisms do not provide support for *programmer-specified* actions to be executed at events.

The semantics of Path Expressions have been defined in terms of the *start* and *term* synchronisation counters [McH89] [Cam83] (the *arrival* counter is not required in the definition). Indeed, it has been proposed that Path Expressions be implemented via these two synchronisation counters [McH89]. Thus, since synchronisation counters are event-based, it follows that Path Expressions are also event-based. However, Path Expressions only embody two of the three events. Like several synchronisation mechanisms that offer programmers direct use of synchronisation counters, Path Expressions do not offer programmers the ability to specify actions to be executed at events.

Finally, synchronisation mechanisms based on Enabled-sets operate by permitting a state transition (from one enabled-set to another) to occur when servicing invocations. Such state transitions are a form of event-based programming, although there is only one state transition/event per operation rather than the three that Sos defines.

We feel that the inability of programmers to specify actions at each of the *arrival*, *start* and *term* events limits expressive power since it precludes the possibility of maintaining synchronisation variables. The utility of actions at these events can be seen from a brief look back at the Esp implementations of synchronisation policies in Section 4.2.

6.3.3 Separation of Synchronisation Code and Data from Sequential Code and Data

Partial separation between synchronisation code/data and sequential code/data can be found in many synchronisation mechanisms.

Synchronisation counters used *by themselves* in guards provide a full separation. However, several guard-based synchronisation mechanisms, such as Guide and DRAGOON, permit guards to access instance variables along with synchronisation counters. Such access to instance variables destroys the strict separation. Not only because synchronisation code is accessing what are *apparently* sequential variables but also because these variables are usually synchronisation variables that happen to be implemented as sequential (instance) variables and the code to maintain these synchronisation variables is mixed in with the sequential code of operations.

Even in modular synchronisation mechanisms that forbid synchronisation code to access instance variables, the separation between synchronisation code and sequential code can be lost. As previously discussed in Section 2.1.4, this often happens if a synchronisation mechanism has limited expressive power and hence needs to combine synchronisation code with sequential code in order to implement some policies beyond its native capabilities.

The Eiffel|| language places its synchronisation code into a single operation, *Live*. In this way, it aims to separate synchronisation code from sequential code. However, here too the separation is not complete since the code in *Live* can access instance variables. What makes Eiffel|| (and other languages that place synchronisation code into a single operation or "body") interesting is that while full separation is not enforced by the language, it can be achieved by programmer convention. For example, instead of implementing synchronisation variables as instance variables, they could be declared as variables local to *Live* and maintained inside this synchronisation operation rather than being maintained inside the bodies of sequential operations.

A lack of separation between synchronisation code and sequential code brings with it several disadvantages.

Firstly, it can be more difficult to read and maintain code that does not separate different types of code from one another.

Secondly, as we will argue in Part III of this thesis, a lack of separation between synchronisation code and sequential code precludes the possibility of language support for generic synchronisation policies.

Finally, as we will shown in Part IV of this thesis, a lack of separation between synchronisation code and sequential code can exacerbate the problems associated with inheritance in COOPLs.

6.3.4 Access to Information about Invocations

It is commonly thought that a synchronisation mechanism needs access to different types of information in order to be expressively powerful. However, we showed in Section 3.6.2 that all this information is available from a single source: the invocation. The SOS paradigm requires that a synchronisation mechanism have access to information about invocations. Actually, the SOS paradigm is rather particular about the kind of access granted and makes a total of five requirements in order to ensure expressive power and modularity.

As already discussed in Section 6.2.3, Mediators satisfies four of the five requirements. (The one requirement Mediators fails to meet is to guarantee that synchronisation code will not access a parameter while it is being updated by sequential code.)

Most other synchronisation mechanisms fare less well against the requirements that Sos makes regarding access to information about invocations.

For instance, many synchronisation mechanisms do not consider invocations to be the primary source of information and hence do not group information about an invocation as a unit. Instead, they provide piecemeal access to some derivative types of information. An example is Path Expressions [CH73] which does not have a construct akin to Esp's *this_inv* for accessing, say, parameters.

The Rosette language implements operation invocations via message passing and each object has its own mailbox to store pending messages. Similarly, the Synchronising Actions (SA) paradigm specifies that operation invocations are implemented via message-passing and objects have their own mailbox to which synchronisation code has access. The mailbox of pending messages meets two requirements of Sos regarding invocations: (i) information about an invocation (message) should be grouped as a unit, and (ii) pending invocations should be held in a collection over which synchronisation code can iterate. However, in papers on Rosette [TS89] and SA [Neu91], there is not a single example illustrating synchronisation code either accessing information contained in a message or iterating over the messages in the mailbox. As such, it appears that the designers of Rosette and SA do not recognise invocations (messages) as being a primary source of information. Also, while Rosette and SA satisfy two Sos requirements regarding access to information about invocations, they do not satisfy all the requirements. For example, they do not provide the ability to declare synchronisation local variables.

The Eiffel|| language provides a class called *REQUEST* that is used to store information about pending invocations. Papers on Eiffel||'s synchronisation mechanism generally do not show synchronisation code *directly* accessing the fields of a *REQUEST*. This is not due to a restriction of access but rather because synchronised classes inherit from the class *PROCESS* which provides a set of utility operations for manipulating the list of pending *REQUESTs*. However, the support that Eiffel|| provides for access to information about invocations is not without limitations. One limitation is that synchronisation code must access parameters not by name but rather by position within an untyped list. As discussed in Section 5.4.3, this approach is error-prone. Another limitation is that there is no facility for programmers to declare their own synchronisation variables local to invocations.

As previously mentioned, the Sos paradigm shows that all the information thought to be required for good expressive power is available from the primary source of invocations. By failing to provide easy and comprehensive access to information about invocations, many synchronisation mechanisms have limited expressive power.

6.4 Contributions of the Sos Paradigm

In this section we summarise the benefits that the Sos paradigm offers.

6.4.1 Safe Access to Instance Variables

As discussed in Section 2.2, it is commonly believed that access to the instance variables of an object by its synchronisation code is needed in order to implement many synchronisation policies. This introduces a problem: synchronisation code must not read an instance variable while that variable is being updated by an operation, otherwise the synchronisation code might see the variable in an inconsistent state.

Searching the literature for synchronisation policies that *apparently* require access to instance variables in order to be implemented unearths policies such as the Bounded Buffer, the Dining Philosophers and the Disk Head Scheduler. We have also introduced another such policy: the Dynamic Priority Print Queue. We have implemented all of these in Section 4.2.3 and in each case the so called "instance" variables in question have turned out to be synchronisation variables. Thus we have backed up our claim, made in Section 3.4, that, in practice, there is no overlap between an object's instance variables and its synchronisation variables. In providing support for synchronisation variables, the Sos paradigm has resolved the problem of synchronisation mechanisms having unsafe access to instance variables.

6.4.2 Expressive Power

In this thesis, the Esp synchronisation mechanism has been used to illustrate the concepts of the Sos paradigm. Through Esp, we have shown that Sos offers excellent expressive power. This has been shown in three different ways.

Firstly, we have shown in Section 4.2.1 that Esp actually subsumes several other synchronisation mechanisms such as synchronisation counters, SP, Path Expressions and the "by" clause of SR. Thus Esp is *at least* as powerful as these mechanisms.

Secondly, the examples in Sections 4.2.2 and 4.2.3 implement many synchronisation policies, including several complex scheduling policies that many other synchronisation mechanisms find difficult to implement, e.g., the Disk Head Scheduler and starvation-free versions of the Dining Philosophers and the Shortest Job Next scheduler.

Finally, Bloom [Blo79] claims that a synchronisation mechanism needs access to six different types of information in order to be expressively powerful. As we discussed in Section 3.6.2, all of these types of information are associated with, or can be derived from, invocations. Since Esp has comprehensive support for accessing and processing information associated with invocations, Esp satisfies all of Bloom's requirements for expressive power.

We warned in Section 2.1 that expressive power often comes at a price that includes one or more of the following: (i) creeping featurism and complexity, (ii) an abrupt degradation of expressive power, and (iii) lack of modularity. None of these prices is paid in Esp, as we now discuss.

Creeping featurism and complexity. As discussed in Section 2.1.1, the expressive power of a synchronisation mechanism can often be increased by introducing a new construct. The desire to have more and more power may result in a synchronisation mechanism containing a mass of constructs. Not only might the sheer number of constructs prove to be a burden for programmers to learn, but the amount of possible interactions between constructs can increase language complexity.

All of the expressive power of the Sos paradigm is derived from just four concepts so Sos cannot be accused of creeping featurism. This is verified in the Esp mechanism which has a remarkably low number of constructs—especially if the syntactic sugar of synchronisation counters and scheduling predicates are removed. The Desp language simplifies matters even further by merging two of the basic concepts of Sos—actions and the means to cause invocations to start execution—with an existing language construct, operations. If the syntactic sugar were removed from Desp then it would have just two language constructs that are not in the sequential language, Dee: (i) the **map** directive, and (ii) the *waiting/executing* expressions.

Degradation of expressive power. It is important for a synchronisation mechanism to exhibit a graceful degradation of expressive power since it assures programmers that the difficulty of implementing a synchronisation policy will be proportional to the complexity of the policy. Often, the expressive power of declarative synchronisation mechanisms is such that they can implement some synchronisation policies in a trivial manner but their ability to implement more complex policies degrades abruptly. On the other hand, procedural mechanisms tend to have less expressive power to start off with but this degrades more gracefully.

As discussed in Section 4.3, Esp is both declarative *and* procedural at heart, and thus permits programmers to choose a mixture of these styles to suit the task at hand. As such, it provides a graceful degradation from declarative into procedural programming.

An excellent example of this is given by the of the Shortest Job Next scheduler. While SP can implement this directly and easily, it fails completely to cope with the slightly more complex, starvation-free variant of the policy. This is a classic example of the expressive power of a declarative mechanism degrading abruptly. However, with the addition of actions in Esp the starvation-free version of the SJN scheduler can be implemented (Figure 4.6 on page 52) in just slightly more work than is required for the less complex version of the policy.

Modularity. It is generally accepted that modularity is very important in sequential programming languages, helping to promote both code readability and maintainability. Modularity is important in concurrent programming languages too. Of particular relevance to this discussion is that, in a COOPL, classes may contain both sequential code and synchronisation code. These serve quite different purposes: sequential code implements the services (operations) that a class provides while synchronisation code ensures that these services can

be provided safely in the face of concurrent invocations.

Because of the difference purposes of sequential code and synchronisation code, they should be kept separate from one another. Certainly, one obvious benefit of such modularity is that it aids code readability: it is easier to read and understand the sequential code of a class if it does not have synchronisation code embedded in it (and vice versa).

The Sos paradigm insists that synchronisation code be separated from sequential code. However, by itself, such a decree is not be enough to guarantee modularity since if a synchronisation mechanism has limited expressive power then programmers might resort to finding a way to work around the decree in order to implement some synchronisation policies beyond the mechanism's expressive power. However, a separation of synchronisation code from sequential code *combined with* a high degree of expressive power makes it possible for Sos to fully segregate synchronisation code from sequential code without placing any apparent limits on the range of synchronisation problems that can be implemented.

Two other benefits of this segregation of synchronisation code and sequential code are as follows. Firstly, this complete segregation is a prerequisite for language support for generic synchronisation policies. This will be discussed in Part III. Secondly, it reduces the problems associated with using inheritance in COOPLs, as we will show in Part IV of this thesis.

6.4.3 Easy Applicability to Languages

Of the four core concepts of the Sos paradigm, not one is new. Each can be found, in one form or another, both in a variety of synchronisation mechanisms and in sequential languages:

- Event-based programming is utilised in sequential programming for tasks such as writing simulations or window-based applications. Events can also be found, albeit implicitly, in a great many synchronisation mechanisms. For example, synchronisation counters, used in many mechanisms, are simply counts of how often particular events have occurred.
- Although different synchronisation mechanisms may differ in what construct they employ to cause invocations to start executing, *every* synchronisation mechanism employs a construct, of some kind, for this purpose.

  The concept of guards—the construct employed in Esp—is used in several synchronisation mechanisms. Guards, albeit with abort rather than delay semantics, are also well-known in sequential programming, e.g., Dijkstra's guarded commands [Dij75] and Eiffel's pre- and post- conditions [Mey92].
- The requirement of the Sos paradigm to separate sequential code and data from synchronisation code and data is a modularity requirement. One natural consequence of it is the introduction of synchronisation variables. Like the other aspects of the Sos paradigm, synchronisation variables are not new. Other synchronisation mechanisms employ synchronisation variables in one form or another, e.g., synchronisation counters. Of course, variables are ubiquitous in sequential programming languages and, as

we discussed in Section 3.5, the introduction of synchronisation variables generalises the existing concept of variables.

- Information about an invocation—notably its parameters and local variables—is accessible within the sequential body of an operation. Usually such information is stored together in an implicit data structure—often termed an *activation record* or a *message*. Likewise, the last core concept of the SOS paradigm groups together similar information about invocations and makes it accessible to synchronisation code.

  As we discussed in Section 3.3, *every* synchronisation mechanism has access to some information about invocations. Many synchronisation mechanisms do not group together information about an invocation in the manner that SOS does. However, *some* synchronisation mechanisms do, e.g., Eiffel|| and Mediators [GC86].

In effect, the SOS paradigm takes several concepts commonly found both in sequential and concurrent programming and combines them in a useful manner. Being composed entirely of existing concepts makes it easy to introduce SOS to a host language since it is likely that some of the core concepts of SOS will already be supported in the language. This was demonstrated with the design of the DESP language in which the concepts of SOS were introduced via just two new language constructs and a handful a keywords.

6.5 Future Work

We bring this chapter, and indeed Part II of this thesis, to a close by briefly outlining some areas for future work.

6.5.1 Optimisation

The performance figures for our prototype implementation of DESP show that an unoptimised implementation of ESP can have a significant run-time overhead. In Section 4.4 we outlined two different optimisation techniques to show that optimisation *is* possible. In Part III of this thesis we will show how generic synchronisation policies reduce the difficulties of optimisation by transformation. However, we have not invested much effort in optimisation and there is still much work that could be done in this area.

6.5.2 Integration with Other Language Constructs

It is well-known that there are problems integrating synchronisation with inheritance. We discuss this issue in detail in Part IV of this thesis. However, there are some other language concepts with which a synchronisation mechanism must also integrate, as we now briefly discuss. A more detailed discussion of the issues involved in integrating synchronisation with these concepts can be found in the "Future Work" chapter (Chapter 14).

Exception handling. If a language supports both concurrency and exceptions then these need to be integrated together. We are not aware of any language which currently does this satisfactorily. One approach would be to extend Sos so that it recognises an *exception* event. For example, the *exception*(Foo) event would occur if execution of operation *Foo* was terminated abnormally. An action associated with such an event could take appropriate measures to ensure that synchronisation variables are left in a consistent state. In effect, just as some languages provide an exception handling mechanism for sequential code, so too might they support exception handling for synchronisation code.

Exception raising. Along with integrating a language's exception *handling* mechanism with its synchronisation mechanism, exception *raising* would also need to be integrated. For example, consider Eiffel-style pre/postconditions [Mey92]. These assertions are expressed in terms of instance variables. Thus the evaluation of a pre/postcondition is similar to the execution of a read-style operation in that it it dangerous to do so if a write-style operation is being executed at the same time. In effect, Eiffel-style pre/postconditions are incompatible with concurrency within an object.

Timeouts. All the events recognised in the Sos paradigm—*arrival*, *start* and *term*—come from a single source: invocations. If other event sources were permitted then this could extend the utility of the paradigm. For example, if timing events were permitted then this would allow *timeouts* to be specified for invocations, which would aid in the development of real-time software.

Part III

Generic Synchronisation Policies

Introduction to Part III

As we discussed in Section 2.3, generic synchronisation policies offer several benefits. For instance, there is the benefit of reuse as a policy need be written only once and can then be instantiated many times. Also, the declarative nature of instantiating a policy can make it easier to use procedural synchronisation mechanisms. Finally, there is the possibility of having skilled programmers write libraries of generic synchronisation policies that can be used by less skilled programmers.

Part III of this thesis is devoted to the topic of generic synchronisation policies. It is structured as follows. We start in Chapter 7 with a discussion of the issues that arise when adding support to a language for generic synchronisation policies. The issues are discussed both in general and, for illustration purposes, in the context of the DESP language (previously introduced in Part II). We show that these issues can be dealt with in a straight-forward manner. The implementation details of a prototype compiler that supports generic synchronisation policies is presented in Chapter 8. Some open issues regarding generic synchronisation policies are discussed in Chapter 9. Chapter 10 brings Part III to a close. It starts off by examining how other languages provide (partial) support for generic synchronisation policies. It then summarizes the contributions we have made in this area and suggests some areas for future work.

Chapter 7

Language Support for Generic Synchronisation Policies

In this chapter, we explore the basic issues involved in providing language support for generic synchronisation policies (GSPs).

This chapter is structured as follows. Section 7.1 outlines the basic language-design issues that need to be addressed. Section 7.2 discusses how the SOS paradigm helps to address some of these issues. To show that language support for generic synchronisation policies is possible, Section 7.3 discusses how generic synchronisation policies can be introduced to the DESP language. This chapter is brought to a close in Section 7.4 with a brief summary of the chapter's main findings.

7.1 Issues to be Addressed

Some of the basic issues that need to be addressed when considering language support for generic synchronisation policies are as follows.

Separation of synchronisation code and sequential code. Generic synchronisation policies will be written outside of the context of any particular sequential class. Thus synchronisation mechanisms that advocate the mixing of synchronisation code with sequential code will not be suitable for writing generic synchronisation policies.

Expressive power. Many existing synchronisation mechanisms currently permit synchronisation code to be mixed with sequential code in an effort to gain more expressive power. This raises the question of whether an enforced separation of synchronisation code from sequential code will have any detrimental effect on the expressive power of a generic synchronisation mechanism.

Different types of formal parameters. Recall some of the sample synchronisation policies introduced in Section 2.3:

policy Mutex[Ops: Set[Operation]]
policy ReadersWriter[ReadOps: Set[Operation], WriteOps: Set[Operation]]
policy BBuf[PutOps: Set[Operation], GetOps: Set[Operation], Size: Int]
policy SJN[Ops: Set[Operation], Length: Parameter]

Notice that all of these are instantiated upon sets of operations. However, one of them, *BBuf*, is also shown to be instantiated upon an integer constant, and another, *SJN*, upon a parameter. This raises the issue of how many *different types* of formal parameter are required in order to support generic synchronisation policies.

Hierarchies of generic synchronisation policies. Some synchronisation policies are similar to one another. This raises the question of whether or not it makes sense to organise generic synchronisation policies into hierarchies analogous to conventional (sequential) class hierarchies.

7.2 Suitability of the Sos Paradigm for Generic Synchronisation Policies

The Sos paradigm proves to be very amenable to the design of mechanisms that support generic synchronisation policies. There are two reasons for this.

Firstly, one of the four basic constituents of the Sos paradigm is the requirement that synchronisation code be separated from sequential code. Furthermore, as can be seen from the Esp mechanism which embodies the Sos paradigm, this separation does not hinder expressive power in any way. Thus the first two issues raised in Section 7.1 are addressed satisfactorily.

Secondly, the Sos paradigm shows that all the information that a synchronisation mechanism requires in order to be expressively powerful comes from one primary source: the invocation. The obvious way to provide a GSP mechanism with information about invocations, and thus ensure the mechanism has good expressive power, is to *instantiate* generic synchronisation policies upon invocations of operations. This is the approach we take. However, before discussing it, we first consider the (lack of) distinction between an *operation* and an *invocation* of an operation.

In considering some programming language constructs, one can see that the distinction between an *operation* and an *invocation* of an operation is not always clear syntactically. For example, consider the following declaration of an operation:

```
Foo(...)
  var x: Int
{ ... }
```

The variable x (syntactically) appears to be local to the *operation* but is actually local to *invocations* of the operation. Similarly, the following guard appears to synchronise operation *Foo* but it actually synchronises *invocations* of operation *Foo*:

Foo: *exec*(Foo) = 0;

In providing language support for generic synchronisation policies, we follow this trend by (syntactically) instantiating generic synchronisation policies upon (sets of) *operations* rather than (sets) *invocations* of operations. In fact, we have already done this in previous examples. For example, consider the *SJN* policy:

policy SJN[Ops: Set[Operation], Length: Parameter]

This syntax suggests that *Ops* are instantiated upon (a set of) *operations* rather than (a set of) *invocations* of operations.

Another point to note about this example is that it treats the parameter *Length* as being syntactically separate from the (invocations of) operations. However, parameters are *part of* invocations and it would be desirable use an alternative syntax to indicate this. One might express this as follows:[1]

policy SJN[Ops: Set[Operation[Length: Int]]]

We said previously that the invocation is the only source of information required in order to have good expressive power. As such, "(sets of) invocations of operations" is the only kind of parameter required by generic synchronisation policies in order to have good expressive power. A practical application of this is that, since there can be no ambiguity as to the type of a formal parameter of a generic synchronisation policy, the syntax used to denote these formal parameters can be shortened by removing the redundant type information. For example, *SJN* policy can be expressed as:

policy SJN[Ops[Length: Int]]

This elimination of redundant syntax has two benefits.

Firstly, it reduces the amount of nested square brackets that appear in declarations of generic synchronisation policies, thus aiding code clarity.

Secondly, the type "Set[Operation]" no longer appears in the syntax, but rather is implicit. This means that language designers are not *obliged* to extend the host language to treat "Operation" as a first class type in order to support generic synchronisation policies. (Of course, language designers may do so if they wish.) Also, the syntax of generic synchronisation policies (GSPs) is now quite different to that of generic/abstract data types (ADTs). This difference in syntax compliments their different semantics. In particular, if there were a combined syntax for both ADTs and GSPs then this might both complicate the writing of compilers and be a source of confusion for programmers.

[1] This is temporary syntax for illustrative purposes only.

The other generic synchronisation policies mentioned previously can be expressed as follows:

```
policy   Mutex[ Ops ]
policy   ReadersWriter[ ReadOps,   WriteOps ]
policy   BBuf[ PutOps,   GetOps,   Init[Size: Int] ]
```

It is common for the size of an object to be passed as a parameter to its constructor. Thus, by instantiating the *Init* parameter of the above *BBuf* policy upon the constructor(s) of a class, the generic synchronisation policy can know the size of the object it is synchronising. A sample instantiation of this can be seen in Figure 7.1.

```
class Buffer {
  ... // instance variables
  Buffer(BufSize: Int) { ... }   // constructor
  Put(...) { ... }
  Get(...) { ... }
synchronisation
  BBuf[ {Put}, {Get}, {Buffer[BufSize]} ];
}
```

Figure 7.1: Bounded Buffer

7.3 Generic Synchronisation Policies in Desp

The discussion in this chapter so far has been somewhat abstract. We now bring the discussion to a more concrete level by discussing what changes need to be made to the Desp language in order to make it support generic synchronisation policies.

7.3.1 Syntax and Usage

We introduce the language changes by example. Consider the bounded buffer class in Figure 7.2. Unfortunately, the syntax "{...}" is already in use in Desp to denote comments, so it cannot be used for sets. We could use "[...]" (as in Pascal) but using this syntax for both sets and genericity would lead to a confusing nesting of square brackets when instantiating generic synchronisation policies. Instead, we use "<...>" to denote sets of operations.

Figure 7.3 shows the implementation of the *BBuf* policy that is instantiated upon the class in Figure 7.2. Note that the synchronisation counters used in this generic policy are expressed in terms of the formal parameters of the policy rather than actual operation names. Similarly, if the generic policy had used *waiting* and *executing* in, say, scheduling predicates, then these would also have been expressed in terms of the policy's formal parameters.

```
class BoundedBuffer[T: Any]
  ... { instance variables }
  public cons make(BufSize: Int) begin ... end    { constructor }
  public method Put(elem: T) begin ... end
  public method Get: T begin ... end
synchronisation
  BBuf[ <Put> <Get> <make[BufSize]> ];
```

Figure 7.2: A bounded buffer class

```
policy BBuf[ PutOps GetOps Init[Size: Int] ]
var Size: Int

method s_Init(t: SizeInv)
begin Size := t.Size; end

method g_PutOps(t: Invocation): Bool
begin
  result := (exec(PutOps GetOps) = 0) and (term(PutOps) - term(GetOps) < Size)
            and (term(Init) > 0);
end

method g_GetOps(t: Invocation): Bool
begin
  result := (exec(PutOps GetOps) = 0) and (term(PutOps) - term(GetOps) > 0)
            and (term(Init) > 0);
end

map start(Init) → s_Init
    guard(PutOps) → g_PutOps
    guard(GetOps) → g_GetOps
```

Figure 7.3: *BBuf* generic synchronisation policy

Note that the guards on *PutOps* and *GetOps* ensure that *Init* has finished executing before permitting these operations to execute. This may seem to be overly cautious since surely programmers would not be foolish enough to concurrently invoke operations upon an object before its constructor has completed execution. (This is technically possible; the creation of the object could take place in one arm of a **cobegin/coend** construct while other arms contain statement to invoke operations upon the object that is being created.) However, it must be remembered that there is nothing in the syntax to restrict the *Init* formal

parameter of the *BBuf* policy to being instantiated upon constructors *only*. It is possible that a programmer might instantiate *Init* upon an operation that is not a constructor in which case it is entirely possible that, say, a *PutOps* operation might be invoked before *Init*.

Note that the *BBuf* policy neglects to ensure that *Init* is invoked only once. The following guard on *Init* could be used to address this concern:

Init: *start*(Init) = 0;

This would have the effect of delaying indefinitely all but the first invocation of *Init*. However, since repeated invocations of *Init* can be considered to be a programming error, it would be preferable to raise an exception rather than delay them indefinitely. Unfortunately, the synchronisation mechanism of DESP has not yet been integrated with its exception handing mechanism so this cannot be done. A discussion of the issues involved in integrating synchronisation mechanisms with exceptions can be found in Section 14.5 of the "Future Work" chapter.

One unexpected benefit that generic synchronisation policies have over writing synchronisation code directly in classes is that type safety has been improved somewhat. Recall from Section 5.4.4 on page 71 that, in DESP, parameters used by synchronisation code are, in effect, declared twice: once in the signature of an operation and a second time as an instance variable in the *Invocation* subclass used for that operation. The compiler checks the names of instance variables in the *Invocation* subclass against the names of parameters in the operation's signature and if there is a match then the compiler generates code to copy the parameter to the corresponding instance variable of the *Invocation* object. The main failing of this scheme is that if a programmer misspells the declaration of the instance variable in the *Invocation* subclass then the compiler assumes it is a synchronisation local variable rather than a misspelt parameter name.

Generic synchronisation policies fix this problem as follows. All of the parameters of an operation that a generic synchronisation policy is interested in are declared in the signature of the policy. The compiler can then check that the *Invocation* subclass associated with an operation contains an instance variable that is a namesake of the formal parameter declared in the policy's signature—and report a compile-time error if this is not the case.

For example, the first line of the *BBuf* policy (Figure 7.3) indicates that *Init* takes a parameter called *Size* of type *Int*. The signature of the *start*(Init) action (operation *s_Init*) informs us that the *Invocation* subclass associated with *Init* is *SizeInv*. The compiler can then check this class (Figure 7.4) to ensure that it has a *Size* instance variable of type *Int*.

7.3.1.1 Inheritance

By default, a subclass inherits the instantiated policy of its parent class. However, a subclass can re-instantiate the generic synchronisation policy used in its parent class (or instantiate a different policy) in order to, say, take account of a new operation. This is illustrated in Figure 7.5, which shows a subclass of the bounded buffer shown in Figure 7.2.

```
class SizeInv inherits Invocation
public var Size: Int

public method set_Size(Val: Int)
begin Size := Val; end
```

Figure 7.4: The *SizeInv* class, used in the *BBuf* policy

```
class ExtendedBuffer[T: Any]  inherits  BoundedBuffer
  public method PutFront(elem: T) begin ... end
synchronisation
  BBuf[ <Put PutFront> <Get> <make[BufSize]> ];
```

Figure 7.5: An extended bounded buffer

7.3.2 Approaches to Implementation

Having used an example in DESP-like syntax to illustrate a generic synchronisation policy and its instantiation, we now discuss possible approaches to code generation.

7.3.2.1 Preprocessing

It would be possible to treat generic synchronisation policies as being macro declarations, e.g., similar to the **#define** preprocessor directive in C [KR78]. In this case the instantiation of a generic synchronisation policy would be treated as the expansion of a (large) macro. The code of the instantiated policy would be placed in-line in the instantiating class by the preprocessor and the preprocessed code passed onto the compiler would be identical to that of the current implementation of DESP.

The advantages and disadvantages of this approach are similar to those of preprocessors in general.

The main advantage is that it would be easier to write a preprocessor to provide a language extension such as this than it would be to modify an existing compiler. However, a disadvantage is that preprocessors tend to be rather simple-minded and do not do any error-checking. Thus programmers would be informed of errors in a generic synchronisation policy only when the policy has been instantiated. As such, the quality of error reporting may be poor.

This problem could potentially be overcome by including syntax and semantic error checking in the preprocessor but this then dramatically increases the complexity of the preprocessor—probably to the point where it would be have been less work to modify the compiler in the first place.

Another problem with the preprocessor approach is that it can lead to code-bloat in applications since it is a form of source code reuse but not of object-code reuse.

7.3.2.2 Treating Generic Synchronisation Policies as Classes

In the discussion so far, the keyword we have used to denote a generic synchronisation policy is **policy**. If this keyword were replaced with **class** then generic synchronisation policies would be very similar (syntactically) to "normal" sequential classes. This suggests that one approach to code generation would be to compile a generic synchronisation policy in a manner similar to the way that sequential classes are compiled. Then, just as instances of a class can be created, so too could instances of a particular generic synchronisation policy.

When a class, *Foo*, instantiates a generic synchronisation policy, *Bar*, the compiler would generate code so that whenever an instance of *Foo* is created at run-time, an instance of *Bar* would be automatically generated to synchronise invocations upon it.

We have modified the DESP compiler so that it supports generic synchronisation, and named this modified language GASP. The approach used in the implementation of this prototype has been to compile policies in a manner similar to the way classes are compiled. We defer discussion of the low-level implementation details until the next chapter so we finish off this discussion by pointing out some of the advantages of proper compiler support for generic synchronisation policies compared to the preprocessor approach.

Firstly, it avoids some of the problems associated with preprocessing that were discussed in Section 7.3.2.1: code bloat and poor error reporting.

Secondly, this approach can lead to some run-time space savings. This is because synchronisation counters need to be allocated only for each formal parameter of a generic synchronisation policy rather than for each synchronised operation in a class. For example, a *ReadersWriter* policy requires that just two sets of synchronisation counters be maintained—one for read-style operations and the other for write-style operations—irrespective of how many operations there actually are in a class that instantiates this policy.

7.3.3 Hierarchies of Generic Synchronisation Policies

As previously said, our GASP prototype treats generic synchronisation policies as being akin to classes. This raises the possibility of having hierarchies of policies. There are two uses for inheritance hierarchies—at least in sequential programming. One is to use inheritance as a means of code reuse. The other is to use hierarchies to indicate subtype (sub-policy) relationships.[2] We discuss each of these possibilities in turn.

Hierarchies for code reuse of generic synchronisation policies. All variants of the readers/writer policy have the following constraints in common:

[2] Note that it is common for object-oriented languages to combine subtyping and code reuse in a single hierarchy, though some people, e.g., Snyder [Sny86, pg. 41], feel that these are separate concepts and languages would be best served by maintaining separate hierarchies for these two purposes. It is outside the scope of this thesis to discuss whether one combined hierarchy or two separate ones are preferable.

ReadOps: *exec*(WriteOps) = 0;
WriteOps: *exec*(ReadOps, WriteOps) = 0;

By themselves, the above guards implement the basic readers/writer policy. Other variants can be derived by **and**-ing on additional constraints. For example, the readers-priority variant can be obtained by **and**-ing on the constraint "*wait*(ReadOps) = 0" to the guard for *WriteOps*. Similarly, the FCFS variant can be obtained by **and**-ing on appropriate scheduling predicates. Thus, one might consider implementing the basic readers/writer policy as a base policy and implementing the other variants as sub-policies that incrementally modify the inherited guards as appropriate.

Since GASP already contained the infrastructure to enable inheritance of classes, it was trivial to extend it to permit inheritance of policies. Our initial experiments in GASP indicate that, in practice, inheritance hierarchies of policies are quite shallow. Thus the ability to inherit generic synchronisation policies appears to be of less importance for code reuse than the ability to instantiate a generic policy multiple times.

The current implementation imposes a restriction on the inheritance of policies: the formal parameter list of a sub-policy must be identical to that of its parent policy. To understand why we imposed this restriction, consider the following policy signatures:

policy Mutex[Ops]
policy FCFS[Ops]
policy ReadersWriter[ReadOps WriteOps]
policy ReadersPriority[ReadOps WriteOps]
policy SJN[Ops[Length: Int]]

Since *Mutex* and *FCFS* has the same formal parameter list, one can be implemented as a sub-policy of the other. Similarly, *ReadersPriority* can be implemented as a sub-policy of *ReadersWriter*, or vice versa. However, the restriction we have imposed means that, say, *Mutex* cannot be a sub-policy of *ReadersWriter*, or vice versa. This seems entirely reasonable for two reasons: (i) they have different numbers of formal parameters, and (ii) the formal parameters have different names. Another possibility would be to permit *SJN* to inherit from *Mutex*. In this case, the two policies both take a single formal parameter, *Ops*. The only difference is that *Ops* in the *SJN* policy takes a parameter, *Length*, while its counterpart in *Mutex* does not. The extra flexibility of permitting related policies to take non-identical formal parameter lists seems slight and thus we decided to disallow it in GASP.

Hierarchies to denote sub-policy relationships of generic synchronisation policies. Inheritance of policies in GASP is used purely as a means of code reuse; there is no concept of one policy semantically being a sub-policy of another. The reason this issue is side-stepped in GASP is that it is beyond the scope of this thesis to define what is meant by saying that one policy is a sub-policy of another.

We are not aware of any work that attempts to define what is meant by one generic synchronisation policy being a sub-policy of another. However, we are aware of some work that deals with subtyping rules for synchronised classes, i.e., classes that contain both sequential code and synchronisation code. Even in this area, the work seems to be of a tentative nature, and the claims of different researchers tend to conflict with each other. For example, Frølund [Frø92] claims that synchronisation constraints in a subclass should be *more restrictive* than those in the base class, while Löhr [Löh91, pg. 21] claims the opposite: that they should be *less restrictive*. Since this topic is outside the scope of this thesis, we do not consider it further.

7.3.4 Optimisation by Transformation

As previously discussed in Section 4.4, one way to optimise Esp is to use a technique known as "optimisation by transformation." Simply put, this involves two steps:

1. Recognise that a piece of synchronisation code implements a particular synchronisation policy.
2. Then replace this synchronisation code with code that implements the same policy in a more efficient manner.

Usually, one might consider the first step—recognising what policy is implemented by a piece of synchronisation code—to be the more difficult of the two. However, it is made almost trivially easy with generic synchronisation policies since the synchronisation policy used is known to the compiler by name. Thus a compiler may recognise the names of several well-known synchronisation policies and special-case the code generation whenever one of them is used.

In order to illustrate that this technique is feasible, the Gasp compiler performs optimisation by transformation for two particular policies: *Mutex* (mutual exclusion) and *FCFS* (a first-come, first-served scheduler).[3] These policies were chosen because of the ease of generating optimised code for them. *Mutex* can be trivially implemented by semaphore operations. In general, one should not assume that processes blocked on a semaphore will be woken up in any particular scheduling order; however, as we implemented the semaphore operations ourselves, we know that blocked processes are woken in a first-come, first-served order. As such, the same optimised code that is generated for *Mutex* can also be generated for *FCFS*.

Obviously, recognising policies by name limits the compiler to being able to optimise only a predefined set of policies—it cannot optimise arbitrary synchronisation code that a user might write. However, it can be a useful stop-gap technique until research has been carried out on recognising policies in arbitrary synchronisation code. Also, it should be noted that although the policies that are optimised in this manner is currently hard-coded into the Gasp

[3] Unoptimised implementations of these policies are also available under the names *UnoptimisedMutex* *UnoptimisedFCFS*.

compiler, this need not be the case. For example, a compiler's configuration file might specify a list of policies and their corresponding optimised object-code files.

One cause for concern with optimisation by transformation is that the compiler blindly assumes that the name of a policy reflects what is actually implemented by the code of the policy. This assumption should be valid for the standard libraries of synchronisation policies shipped with a compiler. However, there is nothing to prevent a programmer from writing their own policy called, say, *Mutex*, that implements something other than mutual exclusion. It would be incorrect for a compiler to optimise this user-written *Mutex* policy into an efficiently implemented mutual exclusion policy. For this reason, the GASP compiler prints a warning message if it applies optimisation by transformation when compiling a policy.

An alternative approach to reduce the danger of incorrectly optimising user-written policies would be for the compiler to apply such optimisation techniques only if, say, the policy being compiled resides in a directory that contains the standard library shipped with the compiler.

7.4 Summary

This chapter has explored the issues that surround the provision of language support for generic synchronisation policies. The main findings of this chapter are as follows.

Firstly, some aspects of the SOS paradigm help in providing language support for GSPs:

- SOS shows that synchronisation code can be completely separated from sequential code without sacrificing expressive power. This is vital since generic synchronisation policies are written independently of sequential code.
- The SOS paradigm shows that all the information a synchronisation mechanism needs in order to have good expressive power can be obtained from one primary source. This results in generic synchronisation policies needing to be instantiated upon just one type of parameter which simplifies the burden of providing language support.

Secondly, generic policies are syntactically similar to classes and can be compiled as such, thus minimising the changes that have to be made to a compiler in order to support GSPs.

Finally, generic synchronisation policies make it easier to implement optimisation by transformation.

In the next chapter we discuss the implementation details of our GASP prototype.

Chapter 8

An Overview of the Prototype Implementation

The previous chapter discussed the issues that arose in providing language support for generic synchronisation policies. In order to illustrate these issues by means of example, the DESP language (originally introduced in Chapter 5) was modified to provide such support. The resulting language was named GASP.

We have implemented a prototype compiler for the GASP language. This prototype acts as a *proof of concept*, i.e., it proves that the concepts of generic synchronisation policies can be implemented without undue difficulty. Since the synchronisation mechanism employed in GASP is ESP, this prototype also acts as a proof of concept for ESP and the SOS paradigm, the concepts of which ESP embodies.

This chapter is structured as follows. Section 8.1 briefly discusses some prior experience of the author that has had an influence on the DESP and GASP prototypes. Section 8.2 discusses the reasons why Dee was chosen as the host language upon which to prototype support for generic synchronisation policies and the ESP synchronisation mechanism. Then in Section 8.3 we give an overview of the architecture of the Dee compiler, linker and run-time system. Section 8.4 discusses an implementation issue that is common to both ESP and generic synchronisation policies. This is followed by discussions of issues specific to the implementation of generic synchronisation policies in Section 8.5, and issues specific to ESP in Section 8.6. Section 8.7 discusses the run-time overhead of synchronisation policies expressed in ESP and how this can be reduced considerably with the aid of optimisation by transformation. Finally, Section 8.8 summarises the main points raised in this chapter.

8.1 Relevant Prior Experience

Two prior experiences of the author have had an influence on the approach taken to implement the current prototype. We mention them here as background material.

Firstly, the author has previously written a tool, called *Pasm*, for teaching concurrency

[McH89]. This involved extending a compiler for a subset of Pascal with the following: (i) a cobegin/coend construct to create processes; (ii) an Ada-like, hierarchical, shared memory address space for processes (child processes have their own private memory segments but also share the address space of common ancestor processes); (iii) semaphores; (iv) monitors; and (v) a debugger with support for these concurrency constructs. That particular compiler produced pcode which was then interpreted. Since the compiler was to be used for teaching purposes, the run-time inefficiency of interpreted pcode was of no concern. Using pcode had the advantage of speeding implementation.

Secondly, in 1991 we implemented a throw-away prototype of the SP synchronisation mechanism. The prototype consisted of writing the run-time support for the synchronisation mechanism as a C**[1] class. Classes to be synchronised inherited from this run-time support class and the guards were "hand compiled" by the programmer. This implementation acted as a proof that the concepts of SP could be implemented; in particular, it allowed us to develop and test the algorithms we had devised for the run-time support of a guard-based mechanism. These same algorithms are used in our DESP and GASP prototypes.

8.2 Choosing a Host Language

Our choice of an object-oriented language upon which to implement ESP and generic synchronisation policies was based purely on pragmatic criteria rather than any features of the language itself. We considered several languages and finally chose Dee [Gro90], for the following two reasons.

Firstly, we had access to the source code of a Dee compiler. This was vital since we would be modifying the language.

Secondly, the Dee compiler produces pcode which is then interpreted. Our previous experience of implementing Pasm and the throw-away prototype of SP had shown us that it was easier to debug the implementation of concurrency constructs in an interpreted pcode environment where one can easily inspect the state of the hypothetical, pcode machine rather than in a compiled environment that utilises a threads package.

A potential disadvantage of this decision is that implementing a synchronisation mechanism on a *hypothetical*, pcode machine does not prove that the mechanism can be implemented on a *real* CPU architecture. In particular, one might devise new pcode instructions in order to facilitate the implementation of new constructs. If such pcode instructions are arbitrarily powerful then it might be difficult to implement similar run-time support on a real CPU that has a fixed instruction set. In this case, the prototype would not be a proof of concept since it would fail to show that the concepts can be implemented on real architectures.

Our approach to this was to assume that a host environment would provide minimal run-time support for concurrency: semaphores and the capability to create processes. We implemented pcode instructions to provide these facilities. All other new pcode instructions

[1] C** [Gro92] is a concurrent, distributed version of C++.

that we devised were no more powerful than instructions found on real CPUs: move values between registers and memory, call and return, etc. In this way, we feel that our prototype illustrates that generic synchronisation policies and Esp are implementable on a real CPU architecture.

8.3 Notes on Dee

Before we discuss the implementation details of the Gasp prototype, we discuss some relevant architectural aspects of the Dee compiler, linker and run-time system.

8.3.1 Hypothetical Machine Architecture

The hypothetical pcode machine [Gro90] is stack-based. There are three stacks.

The *object stack* is used to store parameters and variables declared locally to operations. Actually, this stack holds *pointers* to the objects; the objects themselves are allocated in heap memory.

The *link stack* stores information required by the call/return mechanism used to implement operation invocations. It is common in real CPU architectures for a single stack to combine the purposes of the Dee machine's object stack and link stack. The Dee hypothetical machine maintains separate stacks in order to make garbage collection easier. (The garbage collector knows that *everything* on the object stack is a reference to an object; if the object stack were combined with the link stack then it would have the overhead of determining which entries were object references and which were not.)

The final stack in the Dee hypothetical machine is the *exception stack*. As its name suggests, this holds information necessary to implement the exception handling mechanism of Dee. The integration of synchronisation with exception handling is outside of the the scope of this thesis and so the exception stack will not be considered any further.

8.3.2 Multi-pass Compiler

Dee is implemented by a multi-pass compiler. The first pass parses the entire source code of the class being compiled and a syntax tree representation is constructed.

Then the "interface files" (discussed in Section 8.3.3) of all the classes referenced (either as ancestors or service providers) by the class being compiled are similarly parsed and syntax trees for them are constructed.

The semantic analysis stage traverses and annotates the syntax tree of the class being compiled. When semantic analysis is completed, the annotated syntax tree is again traversed and object-(p)code is generated.

One unusual aspect of the compiler is that it does not maintain a symbol table *per se*. Rather, the syntax trees serve this purpose.

8.3.3 Compiler-generated Files

In order to avoid cluttering the source code directory with extra files, all files produced by the compiler are stored in a sub-directory. In the original Dee compiler, several such sub-directories were used and generated files were stored in these different directories according to purpose. However, we found that setup to be confusing and changed it so that all files generated by the compiler are stored in a single sub-directory, "`./deegen`". Files generated by the compiler include:

- Object-(p)code files for individual classes.
- Pcode "executable" files for linked programs.
- Assembly code versions of the above pcode files.

The compiler also generates "interface files" that describe various interfaces to a class. In particular, it generates:

- A "`.ext`" (extension) file that describes the interface of a class to its heirs. The compiler will read such files in order to determine what instance variables and operations the class being compiled inherits from its parent classes.
- A "`.cli`" (client) file that describes the interface of a class to its clients. In effect, this contains a list of the publicly visible instance variables and operations.

The syntax of these interface files is similar to that of Dee source code files. This permits the same parser to be used for both parsing source code files and interface files.

8.3.4 Offsets for Instance Variables and Operations

When an object is created at run-time, space is allocated from the heap to store the data of the object (header information and instance variables). Instance variables are accessed by their offset within this space allocated for the object. Operations of an object are invoked by means of their offset within the *virtual function table*[2] of the class to which the object belongs.

An unusual aspect of Dee is that these offsets for instance variables and operations are assigned not by the compiler but rather by the linker. The way this works is as follows.

Some of the pcode instructions emitted during the code generation phase of the compiler need to take offsets to *attributes* (instance variables or operations) as arguments. Since the compiler has not allocated offsets for attributes, it instead creates a "dictionary" which is prepended to the object-(p)code file. Each entry in the dictionary is a record which contains

[2] A *virtual function table* is an array of pointers to functions (operations). Each class has its own virtual function table and operations of a class are invoked indirectly by indexing into the class's virtual function table, rather than by invoking an operation directly at the memory address at which the code for that operation is stored. Virtual function tables, which are employed to implement polymorphism, can be found in some other languages including C++ [Str86].

three fields: the *name* of the attribute, the *class* to which it belongs and the attribute's *type* (e.g., whether it is an instance variable or an operation). Whenever a pcode instruction that takes an attribute argument is emitted, the compiler generates a new dictionary entry and passes its location index as the argument to the pcode instruction. In effect, the "dictionary" prepended to the start of the object-(p)code file is relocation information. Other information embedded in an object-(p)code file includes the names of ancestors of the class for which the object-(p)code file is being generated.

The linker reads in all the necessary object-(p)code files and, from the ancestor information embedded in them, constructs an inheritance hierarchy for the entire program being linked. It then traverses this hierarchy to allocate offsets for attributes that are mentioned in the "dictionaries" of the individual object-(p)code files. The algorithm it employs for this task ensures that the offset allocated to an attribute of a particular class will be identical to the offset allocated for the same attribute in all ancestor and descendent classes. Once offsets for all the attributes of classes have been allocated, virtual function tables are created and indexes into dictionaries (that were given as arguments to pcode instructions by the compiler) are replaced by the actual offsets of attributes; finally, the linked executable file is written.

Having the linker assign offsets to the attributes of a class helps to ease the implementation of multiple inheritance. To understand why this is so, consider a class, C, that inherits from two parent classes: A and B. Let us assume that class A has instance variables p, q and r; similarly, assume that class B has instance variables x, y and z. If it were the job of the compiler to assign offsets to the attributes of a class then it might seem natural to start at zero when assigning offsets for the instance variables. Thus, in class A one might expect variable p to be at offset 0, q to be at offset 1 and r to be at offset 2. Similarly, the offsets for class B might be assigned starting from 0. However, this introduces a problem in class C which inherits from A and B since both p and q are at offset 0. The offsets for the other instance variables of A and B will similarly conflict. The problem here is that the offsets for the instance variables of classes A and B were assigned without knowing how these classes might be later combined via multiple inheritance. In general, this information is not known at compile time. However, it *is* known at link time since the Dee linker has knowledge of the *complete* inheritance hierarchy. Thus, the linker could assign offsets for attributes more intelligently. In this example, the linker might assign offsets for the instance variables of class A starting at 0 and the offsets for the instance variables of class B starting at 3. In this way, there is no conflicting assignment of variable offsets in class C.

A disadvantage of this scheme is that it can result in "holes" appearing in the space occupied by instance variables of an object—for instance, offsets 0 to 2 are unused in class B—and in pathological cases this might result in excessive memory wastage. However, it has the advantage of being able to handle multiple inheritance elegantly. Furthermore, in employing this relatively simple algorithm in the linker, the complexity of the compiler is reduced considerably.

8.3.5 Run-time Object Headers

All data types—including "basic" data types such as integers, booleans and strings—are treated as objects. Every object has the following fields in its header.

Kind. An enumerated value specifying if the object is of a standard class that is handled specially (integer, boolean, etc.) or a user-defined class.

Descriptor. A pointer to the class "descriptor" of the object. This is, in effect, a virtual function table.

NextObject. A pointer used by the garbage collector.

Marked. A boolean variables used by the garbage collector to implement a mark and sweep algorithm.

We said in Section 8.3.4 that it is the responsibility of the linker to assign offsets for the instance variables of a class. Since this task is not performed by the compiler, the compiler does not know how much memory space is required to create an instance of a class. Instead, the descriptor of a class (which is created by the linker) contains a short header that specifies the amount of memory that should be allocated for an instance of that class. This class descriptor is passed as an argument to the pcode instruction that allocates memory for an object.

8.3.6 Preparatory Work

In finishing off this discussion about the architecture of the Dee compiler, we ought to mention that we had to extend Dee in some ways that, while not part of the Esp synchronisation mechanism, were necessary in order to support Esp. Notably, a synchronisation mechanism is useless unless there are concurrent processes to be synchronised. Since Dee is a sequential programming language, the concept of multiple processes had to be added. The author's previous experience of implementing Pasm [McH89] was of help in this regard. The cobegin/coend language construct, run-time support for multiple processes and the hierarchical, shared memory address space for processes added to Dee were all based on their counterparts in Pasm.

One other limitation of Dee was the inability to perform "super" calls. It was necessary to add this capability so that we could integrate Esp with a complete inheritance mechanism.

8.4 Implementation Issues Common to both Esp and Generic Synchronisation Policies

Having given an overview of the Dee compiler, we now turn our attention to the implementation of the synchronisation features in Gasp. Although Gasp supports Esp in the form

of generic synchronisation policies, these are relatively independent of one another, and it is possible to have a language that supports Esp but not generic synchronisation policies, or vice versa. As such, it seems sensible to separate the discussion of implementation details specific to generic synchronisation policies from those specific to Esp. However, there is at least one implementation issue that is common to both Esp and generic synchronisation policies. We start off in this section by discussing it. Then in Section 8.5 we consider implementation details specific to generic synchronisation policies. Finally, in Section 8.6 we discuss implementation details specific to Esp.

8.4.1 Synchronisation Wrappers

Consider a class that contains three operations—A, B and C—and instantiates the *Mutex* policy as follows:

```
Mutex[ <A B> ]
```

A and B are said to be *synchronised operations* since they are synchronised by the instantiation of the policy. Conversely, C is said to be an *unsynchronised* operation since it is not mentioned in the instantiation of the policy.

If an operation, *Foo*, of a class is synchronised then the compiler generates code of the form shown in Figure 8.1 for it. Note that the actual sequential code of the operation is surrounded by a *synchronisation wrapper* which is composed of two pieces of code: *pre_sync* and *post_sync*.[3] The *pre_sync* code notes the *arrival* of an invocation, blocks the invocation until its guard becomes true, and then notes the *start* of the invocation. The *post_sync* code notes the *termination* of an invocation. A discussion of the algorithms used in *pre_sync* and *post_sync* will be given later.

Also note that the actual sequential code of *Foo* is placed in a separate operation, *_$_Seq_Foo*, rather than being embedded in-line.[4] This allows for a subclass to, say, incrementally modify (or re-implement anew) the sequential code of *Foo* without the compiler having to generate a new synchronisation wrapper.

[3] This technique of placing a synchronisation wrapper around the sequential code of an operation is commonly employed in the implementation of synchronisation mechanisms. Of course, the actual code generated in the synchronisation wrapper varies from one mechanism to another. Also, there is no standard terminology used to refer to (what we call) *pre_sync* and *post_sync*. For example, Andrews talks of implementing critical regions with "entry and exit protocols" [And91, pg. 98]. Campbell and Habermann implement Path Expressions with a "prologue" and "epilogue" [CH73, pg. 91]. The implementation of Guide's synchronisation mechanism also uses a "prologue" and "epilogue" [DDR+91], while Arche uses a "prelude" and "postlude" [BBI+]. Conditional Path Expressions uses "entrance" and "exit" parts [GW91, pg. 200]. The implementation of Dragoon's behavioural classes uses "request" and "completion" code segments [Atk91, pg. 214].

[4] Note that a dollar sign ("$") cannot appear in an identifier name; thus the "_$_" prefix on this operation precludes the possibility of its name clashing with the names of programmer-declared operations. A similar prefix is used for other compiler-generated, instance variables and operations.

```
Foo(...)
var result: SomeClass;
{
  "pre_sync";
  result := _$_Seq_Foo(...);      // actual code of Foo
  "post_sync";
  return result;
}
```

Figure 8.1: Synchronisation wrapper for a synchronised operation

8.4.2 Unsynchronised Operations

As already discussed, there are two interfaces to a synchronised operation, *Foo*: the actual, sequential code, *_$_Seq_Foo*, and the synchronisation wrapper, *Foo*. A similar, two-interface approach is used for unsynchronised operations. During code generation of an unsynchronised operation the compiler will output the code to *_$_Seq_Foo*; it will then generate a pcode label, *Foo*, that is an alias for *_$_Seq_Foo*.

At run-time, all invocations to an operation, *Foo*, are made through the label *Foo*. For synchronised operations, this invocation will go through the synchronisation wrapper, while if the operation is unsynchronised then the label *Foo* will be an alias for *_$_Seq_Foo* and hence the invocation will be directly on the sequential code.

It may seem redundant to have two identical interfaces for an unsynchronised operation. However, there is a good reason for it. It is possible that an operation will be unsynchronised in the class in which it is first defined—hence the compiler will not generate a synchronisation wrapper for it and *Foo* will be an alias for *_$_Seq_Foo*—but that a subclass will (re-)instantiate a policy and in doing so make *Foo* a synchronised operation. In such cases the compiler will have to generate a synchronisation wrapper for the operation when compiling the subclass. This can be done more easily if the "two-interface" infrastructure for the operation is already in place.

8.5 Implementation Issues Specific to Generic Synchronisation Policies

In this section we discuss aspects of the implementation of Gasp that are specific to its support for generic synchronisation policies. We start off in Section 8.5.1 by discussing the code that is generated for classes that instantiate a generic synchronisation policy. Then in Section 8.5.2 we discuss the code that is generated for generic synchronisation policies themselves.

8.5.1 Code Generation for Classes that Instantiate a Policy

If a class instantiates a generic synchronisation policy then the compiler automatically declares an instance variable, *_$_sync_policy*, for the class. As its name suggests, this variable is a reference to an instance of the policy that is instantiated upon the class.

When an instance of the class is created, an instance of the synchronisation policy instantiated upon it is also created and assigned to the _$_sync_policy instance variable. (Precise details of this are deferred until Section 8.5.1.2.)

8.5.1.1 Synchronisation Wrappers

We mentioned in Section 8.4.1 that a "synchronisation wrapper" is generated for each synchronised operation of a class that instantiates a policy. However, we did not discuss the algorithm used in the implementation of these synchronisation wrappers. We do so now, by means of an example.

The signature of the *ReadersWriter* generic synchronisation policy is as follows:

policy ReadersWriter[ReadOps WriteOps]

Consider a class that instantiates this policy as:

ReadersWriter[<Foo> <Bar>]

The code generated for the synchronisation wrapper of operation *Foo* is shown in Figure 8.2. (The code generated for *Bar* follows a similar template.) As can be seen, the *pre_sync* and *post_sync* of the synchronisation wrapper simply invoke their counterparts in the synchronisation policy. In effect, the synchronisation wrapper delegates to the generic synchronisation policy the responsibility of implementing the appropriate synchronisation code.

```
Foo(...)
var   state: Any;
      result: SomeClass;
{
  state := _$_sync_policy._$_pre_sync_ReadOps();
  result := self._$_Seq_Foo(...);      // actual code of Foo
  _$_sync_policy._$_post_sync_ReadOps(state);
  return result;
}
```

Figure 8.2: Algorithm for synchronisation wrappers in GASP

The name of the *pre_sync* operation of the generic synchronisation policy is constructed automatically by prefixing *_$_pre_sync_* onto the name of the formal parameter of the policy that applies to the operation. For example, operation *Foo* is in the (singleton) set that corresponds to the *ReadOps* formal parameter of the policy so the synchronisation wrapper invokes

_$_pre_sync_ReadOps. The name of the *post_sync* operation that is invoked is constructed in a similar manner.

The *pre_sync* operation of the generic synchronisation policy returns some *state* information which the synchronisation wrapper must pass as a parameter to the *post_sync* operation. The *state* returned by a policy implemented in Esp is an instance of the *Invocation* class denoting *this_inv*. However, a policy implemented in a different mechanism might return back *state* of a different kind.

If a formal parameter of a generic synchronisation policy itself takes parameters then these are passed as parameters to its *pre_sync* operation. For example, let us denote a shortest job next scheduler as follows:

policy SJN[Ops[length: Int]]

In this case, *_$_pre_sync_Ops* operation of the policy takes a single parameter that corresponds to the *length* parameter of *Ops*. As an example of this, consider the class in Figure 8.3 which instantiates the *SJN* policy upon operations *A* and *B* of a class. The synchronisation wrappers generated for *A* and *B* would pass *size* and *len*, respectively, as a parameter to *_$_pre_sync_Ops*.

```
class Foo {
  A(... size: Int ...) { ... }
  B(... len: Int ...) { ... }
synchronisation
  SJN[ <A[size] B[len]> ]
}
```

Figure 8.3: An example of a class that instantiates the *SJN* policy

8.5.1.2 Constructors

We said at the start of Section 8.5.1 that if a class instantiates a policy then the compiler automatically declares an instance variable, *_$_sync_policy* that is a reference to an instance of the policy that is instantiated upon the class. It may seem natural that this variable be initialised by the constructor of the class. However, a class's constructor may itself be one of the operations upon which the policy is instantiated. As such, it is necessary to initialise a synchronised object's policy *before* invoking the constructor for that object—otherwise the synchronisation wrapper of the constructor would try to invoke *pre_sync* upon an uninitialised (instance of the) synchronisation policy.

We use an example to illustrate how this is achieved. Consider the following statement which creates variable *Foo* and invokes its constructor:

new Foo.make(...)

If *Foo*'s class instantiates a generic synchronisation policy then the code generated for the above statement would be as follows:

```
Foo := <allocate memory for the object>
tmp := <allocate memory for an instance of the policy>
tmp.make()      // invoke the constructor for the policy
Foo._$_sync_policy := tmp
Foo.make(...)      // invoke the constructor for the object
```

If *Foo*'s class does not instantiate a generic synchronisation policy then the code in the box would not be generated.

8.5.2 Code Generation for Generic Synchronisation Policies

The GASP language supports the writing of generic synchronisation policies via the ESP synchronisation mechanism. However, the concept of generic synchronisation policies is not restricted to ESP and it is possible to imagine that generic synchronisation policies might be written in a mechanism other than ESP. (In fact, this is already the case since, as discussed Section 7.3.4, the GASP compiler recognises the names of some policies and implements them via semaphores.) As such, this section discusses the requirements that GASP makes on the code generation for generic synchronisation policies. The enforcement of these requirements ensures that there will be a standard object-(p)code interface to a generic synchronisation policy, regardless of how it is implemented. Later, in Section 8.6, we discuss code generation for ESP and show that it conforms to these requirements.

The object-(p)code generated for a generic synchronisation policy must conform to the following two requirements, both of which have already been informally discussed in Section 8.5.1.

The first requirement is that the policy must provide a constructor named *make* that does not take any parameters. As discussed in Section 8.5.1.2, this constructor will be invoked by the run-time system when an instance of the policy is created. This constructor should initialise all the variables that are required by the implementation of the policy.

The second requirement is that for each formal parameter listed in the signature of the policy, there must be a pair of *pre_sync* and *post_sync* operations. As mentioned in Section 8.5.1.1, the name of the *pre_sync* (*post_sync*) operation for a formal parameter is obtained by prefixing *_$_pre_sync_* (*_$_post_sync_*) onto the name of the parameter.

The *pre_sync* operation of a formal parameter, *Foo*, of a policy takes parameters that correspond to the parameters, if any, of *Foo*. It returns a value, denoted as *state*, that can be of any type suitable for the implementation of the policy.

The *post_sync* operation takes a single parameter, which is the *state* returned by its corresponding *pre_sync* operation.

We mentioned in Section 7.3.4 that the current implementation of GASP implements optimisation by transformation for two particular policies: *Mutex* and *FCFS*. The compiler

generates the same code for each of these two policies. The code generated for the *make* constructor initialises a semaphore which the compiler declares as an instance variable of the policy. The code generated for the *presync* operation simply performs a *wait* on this semaphore while the code generated for *postsync* performs a *signal*.

8.6 Implementation Issues Specific to Esp

In this section we discuss aspects of the implementation of Gasp that are specific to its support for Esp. Of course, there is a slight overlap in Gasp's support for Esp and its support for generic synchronisation policies and thus some minor details presented here would be different if Esp were implemented outside of the context of generic synchronisation policies. Such differences are discussed in Section 8.6.5 where we compare the implementation of Gasp with that of Desp (which supports Esp but not generic synchronisation policies).

8.6.1 Run-time Variables Required to Implement Esp

Several variables are needed by the run-time in order to implement a policy written in Esp. These include:

_$_mutex_sem This is a semaphore used to implement a critical region in order to ensure that the synchronisation code contained in a policy is executed in mutual exclusion.

_$_waiting_list This is a list of all the invocations that are currently *waiting* to execute operations.

_$_executing_list This is a list of all the invocations that are currently *executing* operations.

_$_sync_ctrs This is an array of integers used to implement synchronisation counters. The array is big enough to contain a set of three counters for each formal parameter of a generic synchronisation policy.

The compiler inserts these as instance variables of a policy.

As well as the above-mentioned variables associated with a policy, the compiler also inserts some instance variables into the *Invocation* class. These are:

_$_blocking_sem This is a semaphore used to block a process making an invocation until its guard evaluates to true.

_$_op_id This is an integer that indicates the formal parameter of a policy with which the invocation is associated, e.g., for a *ReadersWriter* policy it would indicate if the invocation is a *ReadOps* or *WriteOps*. This variable is used for two purposes.

Firstly, it is it is used as an index into *_$_sync_ctrs* to locate the synchronisation counters associated with a formal parameter, e.g., *ReadOps*, of a policy.

Secondly, as discussed in Section 8.3.4, the linker creates a virtual function table (vtbl) for each (sequential) class. The linker creates *two* vtbls for a synchronisation policy. The first vtbl serves a purpose similar to the vtbl for a class. The second vtbl of a policy specifies which operations of the policy are guards and actions of formal parameters of the policy. (The information to create this second vtbl is obtained from the **map** construct in the source code of a policy.) The *_$_op_id* variable serves as an index into this vtbl to determine what operations to invoke as the guard and actions for an invocation. If a formal parameter does not have, say, an *arrival* action then that entry in the vtbl will be **nil**. if there is a **nil** entry in the vtbl for a guard then a default guard of *true* is used instead.

The algorithms used in the run-time system to maintain these variables will be discussed later.

8.6.2 Code Generation for the Constructor of a Policy

Semantic checks in the GASP compiler ensure that a generic synchronisation policy written by a programmer has a constructor called *make* that does not take any parameters. As discussed in Section 8.5.1.2, when a synchronised object is created, the run-time system will create an instance of the appropriate policy and invoke this constructor. Obviously, programmers can use the constructor to initialise any instance variables they have declared. However, the constructor serves another purpose: the compiler generates code at the start of the constructor to create and initialise the policy's instance variables mentioned in Section 8.6.1 that are required by the run-time system.

The code generated by the compiler is based on the pseudo-code algorithm shown in Figure 8.4. Note that it includes a run-time check to initialise the instance variables required by the run-time system only if they have not already been initialised. This conditional initialisation is necessary since GASP supports inheritance of generic synchronisation policies. As such, it is possible that the constructor of a policy might perform some initialisation and then invoke the constructor of its parent policy. In this case, the constructor of the parent policy should not re-initialise these run-time variables.

8.6.3 Code Generation for *Pre_sync* and *Post_Sync*

In this section we discuss the code generated in the *pre_sync* and *post_sync* operations for formal parameters of a generic synchronisation policy.

Consider the following signature of a generic synchronisation policy:

policy BBuf[PutOps GetOps Init[Size: Int]]

For the purpose of this discussion, we will discuss the code generated for the *pre_sync* and *post_sync* of the *Init* formal parameter. The algorithm of the code generated for *pre_sync* is shown in Figure 8.5.

```
cons make
{ if _$_mutex_sem = nil then
     new _$_mutex_sem.make(1);
     new _$_waiting_list.make_empty_list();
     new _$_executing_list.make_empty_list();
     new _$_sync_ctrs.make(size-of-sync-ctr-array);
  endif
  ...      // programmer-written constructor code
}
```

Figure 8.4: Algorithm used in the constructor of a policy

```
_$_pre_sync_Init(Size: Int)
var this_inv: SizeInv;
{
  this_inv := <allocate memory for a SizeInv>;
  this_inv._$_blocking_sem.init(0);      // set the value of the semaphore to 0
  this_inv._$_op_id := <integer denoting the Init formal parameter>
  this_inv.OpName := "Init";
  this_inv.ClientId := <Process ID of current process>;
  this_inv.arr_time := global_event_clock ++;
  this_inv.Size := Size;      // assign the parameter
  this_inv.make();      // invoke its constructor
  _$_mutex_sem.wait();
    _$_waiting_list.add_at_end(this_inv);
    _$_sync_ctrs[3 * this_inv.op_id + 0] ++;      // arrival(Init) ++
    <invoke the arrival action for this_inv.op_id>
    self.evaluate_guards();
  _$_mutex_sem.signal();
  this_inv._$_blocking_sem.wait();      // block until guard becomes true.
  return this_inv;
}
```

Figure 8.5: Algorithm used in *_$_pre_sync_Init*

The *pre_sync* operation takes a single parameter, *Size*, corresponding to the formal parameter of *Init*. (In contrast, the *pre_sync* operation for, say, *PutOps* does not take any parameters.) The operation declares a variable, *this_inv*. Recall from the discussion of the implementation of *BBuf*, in Section 7.3.1, that the *Invocation* subclass associated with the guard of *Init* is *SizeInv*. Thus, *this_inv* is declared to be of type *SizeInv*.

The first part of this algorithm initialises *this_inv*. Perhaps the only point here that

needs explanation is the initialisation of the *arr_time* of the invocation. This is performed by atomically accessing and incrementing a global event clock. Atomicity for this is obtained by inspecting and incrementing the global event clock by a single pcode instruction. Another approach would have been to declare an event clock as an instance variable of the policy and access it within the critical region of *pre_sync*.

Having initialised *this_inv*, a critical region—implemented via the *_$_mutex_sem* instance variable of the policy—is entered. The first statement of this critical region adds *this_inv* to the list of *waiting* invocations. Then the *arrival*(Init) synchronisation counter is incremented.[5] Following that, the synchronisation virtual function table is examined to determine if there is an *arrival* action registered for *this_inv.op_id*, i.e., for *Init*. If there is, it is invoked, passing *this_inv* as a parameter. The final step in the critical region is to evaluate the guards of all *waiting* invocations. (The algorithm for this is discussed in Section 8.6.3.1.) Having exited the critical region, the process blocks itself until its guard evaluates to true. Finally, it returns *this_inv* to the caller which, as discussed in Section 8.5.1.1, later passes it as a parameter to the *post_sync* operation.

The algorithm for the *post_sync* operation is shown in Figure 8.6. It takes *this_inv* as a parameter. It immediately enters into a critical region, implemented by the same semaphore used for the critical region in *pre_sync*. It removes *this_inv* from the list of *executing* invocations, increments the *term*(Init) synchronisation counter and invokes the *term*(Init) action, if there is one, passing *this_inv* as a parameter. Finally, it evaluates the guards of all *waiting* invocations before exiting the critical region.

```
_$_post_sync_Init(this_inv: SizeInv)
{
  _$_mutex_sem.wait();
    _$_executing_list.remove(this_inv);
    _$_sync_ctrs[3 * this_inv.op_id + 2] ++;      // term(Init) ++
    <invoke the term action for this_inv.op_id>
    self.evaluate_guards();
  _$_mutex_sem.signal();
}
```

Figure 8.6: Algorithm used in *post_sync*

8.6.3.1 Evaluation of Guards

Both *pre_sync* and *post_sync* evaluate the guards of *waiting* invocations. The algorithm used to perform this evaluation is shown in Figure 8.7. In brief, the algorithm iterates over the list

[5] The *arrival*, *start* and *term* synchronisation counters for each formal parameter of the policy are stored in a single array of integers. The run-time system calculates the appropriate index into the array to access a particular counter.

of *waiting* invocations, evaluating the guard of each in turn. If a guard evaluates to true then the following happens: the process blocked on this invocation is woken up and permitted to start execution; the relevant *start* counter is incremented; the invocation is moved from the *waiting* list to the *executing* list; and the relevant *start* action is executed. Furthermore, the evaluation of a guard to true restarts the iteration over the list of *waiting* invocations. This is because code executed at the *start* event updates some synchronisation variables which may, in turn, cause other guards to evaluate to true.

```
evaluate_guards()
var curr_inv: Invocation
    exec_inv: Invocation
    go_again: Boolean
    guard_res: Boolean
{
  go_again := true;
  while go_again do
    go_again := false;
    curr_inv := self._$_waiting_list.head();
    while curr_inv != EndOfList and not go_again do
      if <evaluate guard for curr_inv> then
        exec_inv := curr_inv;
        curr_inv := curr_inv.next();
        exec_inv._$_blocking_sem.signal();       // wake up the invocation
        _$_sync_ctrs[3 * exec_inv._$_op_id + 1] ++;      // exec(...) ++
        _$_waiting_list.remove(exec_inv);
        _$_executing_list.add_at_end(exec_inv);
        <invoke the start action for exec_inv.op_id>
        go_again := true;
      else
        curr_inv := curr_inv.next();
      endif
    end
  end
}
```

Figure 8.7: Algorithm for *evaluate_guards*

8.6.4 Code Generation for Other Synchronisation Constructs

The discussion so far has focussed on code generation and run-time support at the level of entire operations. This section discusses code generation and run-time support for Esp

constructs that are used at the statement level.

8.6.4.1 Synchronisation Counters

Section 8.6.1 mentioned that the *_$_sync_ctrs* instance variable of a policy is an array that stores the values of synchronisation counters. As shown in Figures 8.5, 8.6 and 8.7, the maintenance of these counters takes place in synchronisation wrappers. With such infrastructure in place, the actual accessing of a counter, say, *start*(Foo), is trivial. Code is generated that simply indexes into the synchronisation counter to retrieve the appropriate value. Note that only the *arrival*, *start* and *term* counters are explicitly maintained. The other two counters are evaluated by the run-time system as follows:

$$wait = arrival - start$$
$$exec = start - term$$

8.6.4.2 Scheduling Predicates

As previously discussed in Section 5.6 on page 76, scheduling predicates are simply a syntactic shorthand for a common-or-garden looping construct, and implemented accordingly. A slightly more interesting aspect of compiling a scheduling predicate is the handling of the expression over which the predicate iterates.

Expressions such as *waiting*(Ops) have a type of "Collection[(subtype of) Invocation]". However, the standard Dee class *Collection* is an abstract class that cannot be instantiated directly. As such, the code generated for an expression such as *waiting*(Ops) creates a *List* (which is a concrete subclass of *Collection*). The actual code generated for the expression *waiting*(Ops) implements the algorithm shown in Figure 8.8. Note that this algorithm iterates over the list of *all* pending invocations in order to find those that are *Ops*. As such, it is an $O(N)$ algorithm, where N is the number of pending invocations.

```
list := new List.make();      // this will hold the result of waiting(Ops)
inv := _$_waiting_list.head();
while inv != EndOfList do
  if inv.op_id = <the op_id of formal parameter Ops> then
    list.add(inv);
  endif
  inv := inv.next();
end
```

Figure 8.8: Run-time algorithm to evaluate the expression *waiting*(Ops)

A common case is for, say, the *waiting* expression to refer to *all* the pending invocations. For example, consider the following policy signature:

```
policy SJN[ Ops[Size: Int] ]
```

For such a policy, the expression *waiting*(Ops) refers to *all* pending invocations since *Ops* is the only formal parameter of the policy. In this case, there is no need to make a copy of *_$_waiting_list*: the code generated just returns the list itself.

8.6.5 Comparison with the Implementation of Esp in Desp

Most of the details of the implementation of Esp in Gasp also apply to the implementation of Esp in Desp. However, there are a few small differences that are due to the fact that Desp does not support generic synchronisation policies. In this section we briefly outline these differences.

In Desp, sequential code and synchronisation code are written in the same class. The compiler uses semantic checks to ensure that synchronisation code does not try to access sequential operations or instance variables of the class, and vice versa. The Gasp compiler does not have any need for such semantic checks since synchronisation code is written in a separate class (policy).

In Gasp, synchronisation counters are expressed in terms of the formal parameters of a policy. As such, Gasp allocates a set of synchronisation counters for each formal parameter of a policy. In Desp, synchronisation counters are expressed in terms of operations and hence Desp allocates a set of synchronisation counters for each (sequential) operation defined in a class. This is somewhat wasteful of space, especially since it is possible that not all the sequential operations of a class will be synchronised.

8.7 Run-time Performance

There are two main factors that affect the time spent executing synchronisation code in Gasp.

The first is resource contention. The prototype implementation of Gasp evaluates the guards of all pending invocations at each event. In other words, the number of guards evaluated at an event rises linearly with the number of pending invocations.

The other main factor that affects the amount of time spent executing synchronisation code is the complexity of guards. Put simply, simple guards evaluate quicker than more complex guards, especially guards that contain scheduling predicates.

A less important factor is the presence of actions. This is because an action is executed just once per invocation, unlike a guard which might be evaluated multiple times per invocation.

8.7.1 Measuring Performance

Several programs were written in order to measure the run-time cost of synchronisation code. The programs were all based on the same template: a client object created a number of child processes to make concurrent invocations upon a service object. The programs we used varied from one another in two respects.

The first way the programs varied from one another was in the synchronisation policy that was instantiated upon the service object. In this way we were able to measure how the run-time overhead of synchronisation varied with complexity of synchronisation code. One policy was mutual exclusion, implemented via a simple guard. A second policy also implemented mutual exclusion, but this time by using actions to maintain its own counters and writing the guard in terms of these counters (as in Figure 4.2 on page 48). A third policy implemented a first-come, first-served policy via a scheduling predicate. Finally, we instantiated a first-come, first-served policy that the compiler optimised and transformed into semaphore operations.

The second way test programs varied from one another was in the number of worker processes that were created. In this way, we could measure how synchronisation overhead increased with contention for the service object.

8.7.2 Performance Figures

The graph in Figure 8.9 shows the synchronisation overhead for operation invocation for the different policies previously mentioned.

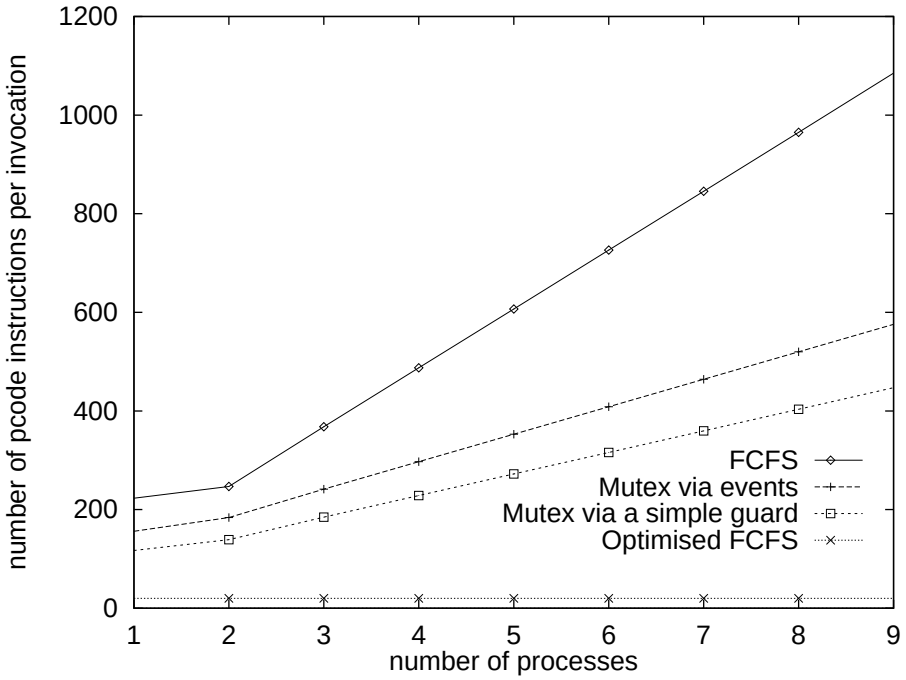

Figure 8.9: Run-time overhead of synchronisation

It should be noted that for the unoptimised synchronisation policies, there is an overhead of six semaphore operations per operation invocation.[6] It is important to note that this overhead is *fixed*, i.e., it does not increase with resource contention. In the prototype

[6]Two semaphore operations are used to implement the critical region for *pre_sync* and another two for *post_sync*. A fifth semaphore operation blocks an invocation until its guard evaluates to true and the sixth is used to wake up this invocation.

implementation, a semaphore operation executes as a single pcode instruction. However, on a real CPU semaphore operations might require hundreds of instructions to implement and thus readers should mentally "shift up" the graphs to take into account a realistic overhead for these six semaphore operations. Since the overhead due to semaphores is fixed, the *slopes* of the graphs will not be affected.

Aside from semaphores, all the pcode instructions used to implement synchronisation code are of comparable power to instructions on a real CPU. Thus, one could expect to obtain similar performance-measuring graphs if DESP were ported to a real CPU. Obviously, the exact figures would be different from one CPU to another but the basic characteristics of the graphs would be similar.

Optimised FCFS policy. As can be seen from bottom-most plot in the graph of Figure 8.9, the synchronisation overhead for the optimised FCFS policy is constant, regardless of resource contention. This is because its implementation via a semaphore avoids the necessity of guard re-evaluation at each event. The actual overhead of this policy is 20 pcode instructions. This includes one instruction to *wait* on a semaphore and another to *signal* the semaphore. The remaining 18 instructions are due to procedure call overhead in the synchronisation wrapper which, as shown in Figure 8.2 on page 120, makes three calls. Two out of these three calls are to invoke *pre_sync* and *post_sync*. The 12 instructions required for these two calls represents the run-time overhead of GSPs. This seems like a small overhead for the benefits that generic synchronisation policies bring. Of course, it is possible to imagine that a compiler or linker might be capable of in-lining the calls to *pre_sync* and *post_sync*, thus avoiding *any* run-time overhead for the use of GSPs. However, the implementation of such a system is outside the scope of this thesis.

The other plots in the graph of Figure 8.9 are for policies that are implemented in ESP and are not optimised. We discuss each of these in turn.

The mutual exclusion policies. Two of the plots in the graph of Figure 8.9 are for mutual exclusion policies.

The lower of these two plots is for the policy implemented via a simple guard of the form:

Ops: $exec(\text{Ops}) = 0$;

As previously stated, the number of guard re-evaluations at each event is proportional to the number of pending invocations. This results in a linear rise in the cost of synchronisation, as can be seen from the plot.

The other implementation of mutual exclusion plotted in Figure 8.9 uses events to maintain, what are in effect, synchronisation counters and these are then used in a guard. (The algorithm for this was shown previously in Figure 4.2 on page 48.) The presence of the actions results in a higher initial overhead for the implementation of the policy. This overhead works out to be 11 pcode instructions for each of the three actions. (The figure of 11 instructions

consists of 4 instructions to actually invoke an action and 7 instructions to execute the body of the action.) The overhead of these actions is constant, i.e., it does not affect the slope of the graph. From looking at the graph it may appear that the actions add a relatively large amount of overhead. However, this is not the case since the run-time overhead of the synchronisation mechanism, irrespective of whether or not actions are used, includes six semaphore operations. If the plots were shifted up to reflect the true cost of these semaphore operations then it would be apparent that the additional overhead of using actions is negligible. This means that a synchronisation mechanism that employs guards can also provide programmers with the ability to execute actions at events—thereby increasing the mechanism's expressive power—at very little extra cost.

The guard of this mutual exclusion policy is expressed in terms of the programmer-maintained counters. This takes slightly longer to evaluate than a similar guard expressed in term of the automatically-maintained *exec* counter; hence the slight difference in slopes for the plots of synchronisation overhead for the two implementations of the mutual exclusion policy.

Unoptimised FCFS policy. The top-most plot in the graph of Figure 8.9 is for the first-come, first-served policy implemented via a scheduling predicate of the form:

```
Ops: exec(Ops) = 0
     and there_is_no(f in waiting(Ops): f.arr_time < this_inv.arr_time);
```

As can be seen, the presence of a scheduling predicate adds significantly to the cost of synchronisation. One might expect that the cost of including a scheduling predicate in a guard would be $O(N^2)$, where N is the number of *waiting* invocations. However, most times that the guard is evaluated, the conjunct "*exec*(Ops) = 0" will be false and this will "short-circuit" the condition's evaluation, thus avoiding evaluation of the scheduling predicate. For this FCFS policy, this "short-circuit" evaluation results in the predicate being evaluated only once per invocation rather than N times at each event. Thus, for this particular policy the presence of a scheduling predicate does not result in $O(N^2)$ overhead for synchronisation. However, it may for other policies.

8.7.3 Discussion

The plots in the graph of Figure 8.9 show that, in an unoptimised implementation of Esp, the synchronisation overhead for an invocation upon a synchronised operation is in the region of hundreds or thousands of instructions. This overhead precludes the use of an unoptimised implementation of Esp in fine-grained concurrent applications. However, this synchronisation overhead may be acceptable for applications that employ concurrency at a coarser granularity.

If usage of Esp is not to be restricted to coarse-grained concurrent systems then optimisation techniques become of some importance. Although we briefly outlined some optimisation techniques in Section 4.4 and implemented a sample of optimisation by transformation to

prove its feasibility, the issue of optimisation is one that has not been addressed in depth in this thesis.

While optimisation for Esp may be difficult, the technique of optimisation by transformation seems ideally suited for generic synchronisation policies. This suggests that in a language to be used for writing fine-grained concurrent applications, generic synchronisation policies may be more important than an expressively powerful synchronisation mechanism such as Esp.

It is important to note that the synchronisation overhead is for invocations upon synchronised operations only. The introduction of synchronisation into the host language, Dee, has not introduced any execution-time overhead for invocations upon unsynchronised operations.

Similarly, instances of classes that do not instantiate a synchronisation policy do not pay any space overhead since the compiler inserts the instance variables required for the implementation of Esp and generic synchronisation policies only in those classes where they are required. The only space overhead paid for by all classes, regardless of whether or not they are synchronised, is that the virtual function tables are larger, due to there being two entry points for each operation (as discussed in Section 8.4.2). This will result in a slight increase in the size of executable files but no per-object space or time overhead.

8.7.4 Related Work

It seems that very few designers of synchronisation mechanisms provide details of the run-time overhead of synchronisation in papers that they write.

One notable exception is Matsuoka *et al.* [Mat93] [MTY93] [TM93] who provide details of the run-time overhead of synchronisation in their implementation of the ABCL language on a massively parallel computer. ABCL supports concurrency between objects but not within an object. This simplifies their task of implementation somewhat, but even so they have achieved very impressive results: a synchronised invocation takes just a dozen or so CPU instructions more than an unsynchronised call.

One interesting optimisation technique used in ABCL is for the compiler to generate several virtual function tables for a synchronised class. As a simplified example, consider a bounded buffer that can be in one of the following states: *empty*, *partial* or *full*. Whether or not a particular operation on a bounded buffer will execute immediately or will be delayed depends on the current state of the buffer. Thus, the entry for the *Put* operation in the virtual function tables for the *empty* and *partial* states might point directly to the sequential code that implements *Put*, while the entry in the table for the *full* state would point to a routine that queues the invocation for later processing. This use of multiple virtual function tables integrates some of the synchronisation overhead with the standard vtbl-lookup which is done for an invocation, regardless of whether it is synchronised or not.

Bergmans [Ber94] uses a technique similar in purpose to the re-evaluation matrix (discussed in Section 4.4.1) to reduce how often guards have to be evaluated. With the profiling tools available to him, Bergmans was able to measure the reduction in total execution time

of test programs but not the reduction in time spent executing synchronisation code. As such, it is known that the optimisation technique in question works but it is not known how effective it is.

8.8 Summary

This chapter has discussed the implementation of both ESP and generic synchronisation policies in our GASP prototype.

We have shown that both ESP and GSPs can be implemented in a straight-forward manner.

Our performance analysis of ESP shows that the run-time overhead of synchronisation is hundreds or thousands of instructions per invocation of an synchronised operation. This suggests the need for optimisation, which is an issue that this thesis does not explore in depth.

We have shown that the overhead of using generic synchronisation policies in preference to writing synchronisation code directly in a class is about a dozen instructions per operation invocation, which is a relatively small overhead compared to the overhead of ESP. Furthermore, generic synchronisation policies can be used to implement optimisation by transformation which can reduce the overhead of ESP significantly.

Chapter 9

Open Issues for Generic Synchronisation Policies

Chapter 7 addressed basic issues concerning language support for generic synchronisation policies and Chapter 8 discussed the implementation of the GASP prototype which provides such support. In this chapter we discuss some issues regarding generic synchronisation policies that have not been addressed in our GASP prototype.

This chapter is structured as follows. We start in Section 9.1 by discussing how defining a standard, object-code interface to generic synchronisation policies raises some interesting possibilities. Then in Section 9.2 we discuss the possibility of a subclass being able to incrementally modify the instantiation of a synchronisation policy in its parent class. Finally, Section 9.3 raises the possibility of being able to instantiate several policies upon the operations of a class.

9.1 A Standard, Object-code Interface to Generic Synchronisation Policies

In Section 8.5.2 we said that the GASP compiler defines an object-code interface to generic synchronisation policies. If this, (or a similar) object-code interface were adopted as part of a language standard then it would mean that programmers would have complete freedom in how they implemented a generic synchronisation policy (as long as its compiled code conformed to the standard, object-code interface). In particular, a programmer could implement a policy in a synchronisation mechanism of choice.

One benefit of having a standard, object-code interface to policies has already been addressed in previous chapters: it makes it possible for the compiler to perform optimisation by transformation. For example, it makes no difference to a class that instantiates, say, a *Mutex* policy whether that policy is implemented as a guard or as semaphore operations.

Another possibility is that several synchronisation mechanisms might be supported at the language level. Of course, a language might do this without the aid of generic synchronisation

policies; however, GSPs make this more feasible by completely separating synchronisation code from sequential code and thus preventing interaction between different synchronisation mechanisms and sequential code which in turn keeps language complexity down.

A potential advantage of supporting several synchronisation mechanisms in a single language is that it enables programmers to choose whichever mechanism they want to use. For example, a programmer might initially implement a complex scheduling policy in a high-level, but inefficient synchronisation mechanism, e.g., an unoptimised version of Esp. If later, during execution-time profiling of an application, it is discovered that the a substantial amount of time is being spent executing the code of this policy then the programmer may decide to re-implement it in a lower-level, and more efficient, synchronisation mechanism.

Instead of supporting several synchronisation mechanisms, a language might support *none* and force programmers to fulfill all of their synchronisation requirements by instantiating policies that have been implemented in a different language. The advantage of this is that, by not supporting any synchronisation mechanism, the language is kept as small as possible which simplifies the task of compiler writers. In particular, it is likely to be easier to extend an existing language to enable it to instantiate generic synchronisation policies than it is to extend the same language with a synchronisation mechanism.

Of course, if a general-purpose programming language can instantiate generic synchronisation policies but cannot be used to write them then a different language is needed to actually implement GSPs. Since this latter language is quite specialised (it might be used *only* to write synchronisation policies) it is likely be smaller and easier to implement than a general-purpose language. For example, Path Expressions [CH73] is usually considered to be a synchronisation mechanism; However, it could be considered to be a complete, specialised language used to implement generic synchronisation policies.

A final idea is that future operating systems might provide efficient implementations of commonly used synchronisation policies as system services. These system services could then be encapsulated as generic synchronisation policies for use by programmers.

9.2 Incremental Modification of Inherited, Instantiated Policies

In Section 7.3.1.1 we mentioned that, by default, a subclass in Gasp inherits the instantiated policy "as is" from its parent class, but that it is free to re-instantiate the same policy (or instantiate a different policy). In this section, we discuss another possibility, albeit one that is not supported in the current implementation of Gasp.

Often the instantiation of a generic synchronisation policy in a subclass will be similar to the instantiation of a policy in its parent class. In such cases it would be useful to be able to express the instantiation of a policy in a subclass as an incremental modification of the policy instantiation in the parent class.

For example, consider a class that instantiates the *ReadersWriter* policy as follows:

ReadersWriter[{A, B}, {C}]

If a subclass introduces a write-style operation, D, then it might instantiate the *ReadersWriter* policy as follows:

ReadersWriter[super.ReadOps, super.WriteOps + {D}]

Here we see that the first actual parameter to the subclass's synchronisation policy is inherited "as is" from the corresponding parameter to the synchronisation policy of its parent class. The second actual parameter is inherited and incrementally modified by means of the "+" (set union) operator.

As another example to illustrate the incremental modification of the instantiation of a generic synchronisation policy, consider a subclass that, instead of introducing a new operation, re-implements operation B and in doing so makes it a write-style operation. The subclass might express this change as follows:

ReadersWriter[super.ReadOps − {B}, super.WriteOps + {B}]

The use of "super" need not be confined to the actual parameters of an instantiated policy; it might also be used to refer to the actual policy instantiated. For example, the previous example could have been written as:

super.policy[super.ReadOps − {B}, super.WriteOps + {B}]

If, sometime later, a programmer modified the parent class so that it instantiated, say, the *ReadersPriority* rather than the *ReadersWriter* policy then this change would automatically be reflected in the subclass. Of course, this only works because the *ReadersWriter* and *ReadersPriority* policies both use the same names for their formal parameters. If a programmer changed the policy in the parent class to be, say, *Mutex*, then the subclass would no longer compile because the expression "super.ReadOps" would no longer make sense.

9.3 Instantiating Several Policies on the Operations of a Class

It is likely that some classes will have idiosyncratic synchronisation requirements that cannot be handled by already-written generic synchronisation policies. For example, a bounded buffer policy, *BBuf*, might be instantiated upon the following sets of operations: *PutOps*, *GetOps* and *Init* (a constructor that takes the size of the buffer as a parameter). However, a particular bounded buffer class might also contain other operations that do not fit neatly into these categories, e.g., operations that return the state of the buffer such as *IsEmpty* and *IsFull*.

One way to tackle classes with idiosyncratic synchronisation requirements would be to write new generic synchronisation policies that are tailored to their exact needs. However, there are two problems with this approach.

Firstly, creating several variations on policies such as *BBuf*, *Mutex*, *ReadersWriter* and *SJN* would quickly lead to name-space pollution.

Secondly, this approach goes against the spirit of genericity since many "generic" policies would be written for a *specific* class and would probably be instantiated upon *only* that class.

We feel that a better approach to the problem of classes that have idiosyncratic synchronisation requirements is to provide language support that allows programmers to instantiate *several* policies upon the operations of a class. In this way, existing generic synchronisation policies can be combined in order to effect variations of them. We illustrate the concept in Section 9.3.1 through some examples and then in Section 9.3.2 we briefly discuss some of the problems that need to be addressed in order to be able to support this ability of instantiating several policies upon a single class.

9.3.1 Examples

In this section we present some examples to illustrate how it can sometimes be useful to instantiate several generic synchronisation policies on a single class in order to combine them to better suit the synchronisation needs of the class.

Overlapping *ReadersWriter* Policies

Consider the sequential code of the class on the left in Figure 9.1. Since operation A examines (reads) instance variables and the other two operations, B and C, update (write) them, one might consider that a suitable synchronisation policy for this class would be:

ReadersWriter[$\{A\}$, $\{B, C\}$]

However, this is needlessly restrictive since the write-style operations, B and C, update *different* instance variables—hence there is no reason why the execution of one should restrict the execution of the other.

```
class Foo {
  int  x, y;      // instance variables
  A(...) { ...; "examine x & y"; ... }
  B(...) { ...; "update x"; ... }
  C(...) { ...; "update y"; ... }
synchronisation
  ReadersWriter[ {A}, {B, C} ];
}
```

```
class Foo {
  int  x, y;      // instance variables
  A(...) { ...; "examine x & y"; ... }
  B(...) { ...; "update x"; ... }
  C(...) { ...; "update y"; ... }
synchronisation
  ReadersWriter[ {A}, {B} ];
  ReadersWriter[ {A}, {C} ];
}
```

Figure 9.1: Right: a class with an overly restrictive synchronisation policy. Left: Instantiating *ReadersWriter* multiple times in order to remove the restriction.

In order to remove this unnecessary restriction on concurrency, one might consider each instance variable in turn and note the manner in which it is accessed by operations. In this case, we note that variable x is examined by A and updated by B. Thus the following policy would be suitable to protect x alone:

ReadersWriter[$\{A\}$, $\{B\}$] /1/

Similarly, variable y is examined by A and updated by C. Thus, the following policy would be suitable to protect y:

ReadersWriter[$\{A\}$, $\{C\}$] /2/

With the ability to instantiate multiple synchronisation policies on a class, these two instantiated policies might be combined as shown in the class on the right in Figure 9.1.

Note that the semantics of combining synchronisation policies implemented as guards is quite intuitive: simply **and** the guards together. To see this, consider that policy /1/ might be expressed as the following guards:

A: *exec*(B) = 0;
B: *exec*(A, B) = 0;

Similarly, policy /2/ might be expressed as the following guards:

A: *exec*(C) = 0;
C: *exec*(A, C) = 0;

Combining these two sets of guards would yield:

A: *exec*(B) = 0 **and** *exec*(C) = 0;
B: *exec*(A, B) = 0;
C: *exec*(A, C) = 0;

Combining *BBuf* with *ReadersWriter*

One usually thinks of a bounded buffer containing operations *Put* and *Get*, and perhaps a constructor that takes a parameter indicating the size of the buffer. A generic synchronisation policy, say, *BBuf*, might be written that is suitable for such a class. However, if a bounded buffer class also contains other operations, e.g., *IsEmpty* and *IsFull*, then the *BBuf* policy would not be suitable for the class. The class in Figure 9.2 illustrates one way to tackle this problem. It instantiates the *BBuf* policy but also instantiates *ReadersWriter* to prevent *IsEmpty* and *IsFull* executing while the buffer is being updated by *Put* and *Get*. The rule of **and**-ing together guards works fine in this example.

```
class Buffer {
  ...   // instance variables
  Buffer(int Size) { ... }   // constructor
  Put(...) { ... }
  Get(...) { ... }
  IsEmpty(...) { ... }
  IsFull(...) { ... }
synchronisation
  Bbuf[ {Put}, {Get}, {Buffer[Size]} ];
  ReadersWriter[ {IsEmpty, IsFull}, {Put, Get} ];
}
```

Figure 9.2: A bounded buffer that combines the *BBuf* and *ReadersWriter* policies

Instantiating *Mutex* Multiple Times on a Class

Consider a class that contains four operations—A, B, C and D—each of which updates instance variables. One might consider employing the following policy upon this class:

Mutex[{A, B, C, D}]

However, if the operations update disjoint sets of instance variables then this policy would be overly restrictive. Instead, one might instantiate *Mutex* several times, as shown in Figure 9.3.

```
class Foo {
  ...   // instance variables
  A(...) { ... }     B(...) { ... }     C(...) { ... }     D(...) { ... }
synchronisation
  Mutex[ {A} ];
  Mutex[ {B} ];
  Mutex[ {C} ];
  Mutex[ {D} ];
}
```

Figure 9.3: A class that instantiates the *Mutex* policy multiple times

9.3.2 Issues to be Addressed

The examples in Section 9.3.1 illustrate the utility of combining generic synchronisation policies by multiple instantiations on a class. However, until there has been extensive usage of GSPs in practice, it is difficult to know for sure just how important this capability is. One thing is certain, however; in order to provide support for such multiple instantiations, some issues need to be addressed. This section briefly outlines what these issues are.

Defining the semantics of combining multiple policies In Section 9.3.1 we said that the semantics of combining synchronisation policies expressed as guards is quite intuitive: simply **and** the guards together. However, unless a GSP-based synchronisation mechanism is to be restricted to consist of guards alone then the semantics of combining other synchronisation constructs needs to be defined too.

For example, Esp code can consist of actions and programmer-maintained synchronisation variables as well as guards so semantics need to be defined for the combining of actions and variables from different policies. A first attempt at defining suitable semantics might be as follows:

- Keep the synchronisation variables of different policies separate from one another. This is necessary in order to avoid conflicts, e.g., one policy might have a synchronisation variable, *foo*, of type *Int* while another policy might define a variable of the same name to be of type *Bool*.
- At a particular event, execute (in an unspecified order) the actions, if any, that the different policies specify for that event.
- An invocation cannot *start* execution until the guards specified for it by *all* the different policies evaluate to true, i.e., the effective guard for an invocation is the **and**-ing together of all the guards from different policies for that invocation.

With each policy having its own synchronisation variables, the policies, in effect, run in step with one another. It appears that this will lead to intuitive semantics for combining policies but we have not considered this at length to determine if this is, in fact, the case.

It should be noted that separation of synchronisation variables of different policies produces a difficulty: each policy would have to maintain its own lists of *waiting* and *executing* invocations. Storing multiple copies of invocations is certainly wasteful of memory. It may also lead to extra run-time overhead in determining which invocations from different policies correspond to one another.

It should also be noted that the ability to combine several policies will probably require an approach to implementation that is more complex than that required for non-combined policies.

Semantic subtleties of combining multiple policies. Another difficulty with the use of multiple generic synchronisation policies is the subtle ways in which different policies might interact with each other and result in deadlock. For example, consider the class in Figure 9.4 which instantiates both a first-come, first-served (FCFS) policy and a Shortest Job Next scheduler upon operation *A*. These two policies conflict with each other in scheduling invocations for execution and will cause deadlock if jobs do not happen to arrive in order of increasing *length*. Corradi and Leonardi also warn of the dangers of combining several incompatible constraints [CL91, pg. 50].

```
class Foo {
  A(int length, ...) { ... }
synchronisation
  FCFS[ {A} ];
  SJN[ {A[length]} ];
}
```

Figure 9.4: Instantiating conflicting scheduling policies upon a class

Another example of inadvertent deadlock caused by combining incompatible synchronisation policies can be found in papers on the VCP system [CGM91, GC92]. One example given in these papers to illustrate how VCP's synchronisation mechanism interacts with inheritance is as follows.

A bounded buffer class is written which includes guards of the form:

Put: $exec(\text{Put}) = 0$ **and** $term(\text{Put}) - term(\text{Get}) < \text{Size}$;
Get: $exec(\text{Get}) = 0$ **and** $term(\text{Put}) - term(\text{Get}) > 0$;

A subclass of this bounded buffer modifies the synchronisation policy to give priority to *Get* invocations over those of *Put*. Towards this goal, the constraint "$wait(\text{Get}) = 0$" is added to the guard of *Put*. Unfortunately, this new constraint causes deadlock. To see why, consider a buffer that is currently empty and has some pending *Get* invocations. If a *Put* invocation arrives then it cannot proceed due to these pending invocations. Conversely the pending *Get* invocations cannot proceed until the *Put* invocation executes and thus makes the buffer non-empty. With neither *Put* nor *Get* able to execute, the buffer is deadlocked.

In VCP, guards are not written in the form we have shown but rather individual constraints are given names (e.g., *mutex* or *priority*) and these are then combined to form what are, in effect, guards. Although VCP's named constraints are not generic in nature, they are similar to generic synchronisation policies in that they provide a layer of abstraction.

The authors of the VCP papers claim that such abstraction helps in the writing of complex guards. This claim is based on an example which uses VCP's abstraction mechanism to combine three different variations of a bounded buffer policy, one of which is the deadlocking get-priority policy already discussed. The authors claim that compared to the abstractions that are used, the equivalent guards "are surprisingly complex and, surely, difficult to directly design and reuse" [GC92, pg. 5]. The irony is that the authors make such a statement while being unaware of the deadlocking bug in their own code. This clearly illustrates that modifying synchronisation policies (whether by incremental modification during inheritance or by combining several abstract/generic policies together) can result in unforeseen, and unpleasant, semantic subtleties.

9.4 Summary

This chapter has discussed some issues concerning generic synchronisation policies that are not addressed in the current implementation of GASP.

One limitation of GASP is that it does not permit a subclass to incrementally modify the instantiated policy in its parent class. Instead, a subclass must either inherit its parent class's instantiated policy "as is" or re-instantiate it anew. We shall return to this subject in Part IV which focuses on the issue of inheritance in COOPLs.

Another limitation is that GASP does nor provide programmers with the ability to instantiate several policies upon the operations of a class. We feel that such a capability would be useful but recognise that it may be difficult to implement.

A third topic discussed in this chapter was not a limitation of GASP, but rather the interesting possibilities that are presented by having a standard, object-code interface to generic synchronisation policies.

Chapter 10

Related Work and Summary of Contributions

Part III of this thesis has focussed on the feasibility of providing language support for generic synchronisation policies. This chapter brings Part III to a close. We start off in Section 10.1 by discussing how several other languages attempt to support generic synchronisation policies. Then in Section 10.2 we summarise own own contributions to this area. Finally, in Section 10.3 we recap on some limitations of our work and suggest areas for future research.

10.1 Related Work

We know of four languages, other than GASP, that provide support, in one form or another, for generic synchronisation policies: DRAGOON, Demeter, HECTOR and Parallel Objects. Unfortunately, all of these languages have limitations in the support that they offer. These limitations are discussed in Sections 10.1.1 through to 10.1.4.

Writing synchronisation code in terms of abstract sets of operations is central to generic synchronisation policies. Abstract sets of operations are also employed in the Enabled-sets synchronisation mechanism. This raises the question whether Enabled-sets would be a suitable basis for supporting GSPs? This issue is discussed in Section 10.1.5.

10.1.1 Dragoon

Next to GASP, the language which provides the best support for GSPs is probably DRAGOON [Atk90]. However, DRAGOON's support for generic synchronisation policies has some limitations as we now discuss.

10.1.1.1 Confusion of Inheritance and Genericity

In DRAGOON, generic synchronisation policies are referred to as *behavioural classes*. Unfortunately, it appears that the DRAGOON language designers were not fully aware of the generic

nature of behavioural classes when originally designing the language. Some examples that illustrate this lack of appreciation for the generic nature of behavioural classes are as follows:

- From the earliest papers published on DRAGOON[MCB+89, GMC+89], the application of a behavioural class upon a sequential class has been consistently referred to by the term *behavioural inheritance* which is said to be "a form of multiple inheritance" [Atk90, pg. 103]. The fact that these papers talked about the *inheritance* of behavioural classes rather than their *instantiation* suggests that the language designers had confused the concepts of genericity and inheritance.
- Originally the only formal parameter type for behavioural classes were sets of operation names. However, in an attempt to increase expressive power, some proposed extensions to DRAGOON's synchronisation mechanism [RdPG91] permitted other types of formal parameters to be specified, such as integer constants and functions that returned the state of the object being synchronised. The keyword **generic** was also introduced. However, this keyword was used *only* with behavioural classes that contained one or more of the new formal parameter types. In particular, the **generic** keyword was not retro-fitted to the original style of behavioural class which indicates that the language designers did not consider the original style of behavioural class to be generic.[1]

This confusion regarding the generic nature of behavioural classes has left its mark on the DRAGOON language. In particular, the use of "behavioural inheritance" to instantiate generic policies has several disadvantages, which we now discuss.

Inconvenience for programmers. There are several practical ways in which behavioural inheritance is less convenient for programmers than the ability to instantiate generic synchronisation policies in-line as we have advocated for DESP:

- Firstly, the only way to create a synchronised class is to create a sequential version, say, class *Person*, and then use behavioural inheritance to create the desired synchronised class, say, *MutexPerson*. Programmers have to create *both* of these classes even if they have no need for the sequential class. This can result in name-space pollution of classes.
- Secondly, there is more syntactic baggage, and hence typing, involved in behavioural inheritance. This may seem like a petty issue but it can be disheartening to programmers when the amount of typing it takes to instantiate a generic synchronisation policy is often as long as the implementation of the synchronisation policy itself.

Language inconsistencies. Aside from practical issues, behavioural inheritance has introduced inconsistencies into the language. In particular, programmers cannot inherit from a "behavioured class", i.e., a class that is the result of behavioural inheritance. This can result

[1] More recently, the core concepts of behavioural classes have appeared, under different syntax, as a proposal for Ada9X [AW93]. The syntax of this new proposal indicates all the formal parameter types as being generic.

in idiosyncrasies in the type system of the language. For example, *Student* may be a subclass of *Person* but *MutexStudent* cannot be a subclass of *MutexPerson*.

10.1.1.2 Limited Expressive Power

Originally, DRAGOON's behavioural classes took just a single type of formal parameter: sets of operation *names*; no other information about invocations of operations was available, e.g., relative arrival times, parameters or synchronisation local variables. Because of this, DRAGOON's synchronisation mechanism had limited expressive power.

Proposed extensions [RdPG91] introduce additional types of formal parameters and *five* new constructs to compliment the original two (synchronisation counters and access to instance variables via functions) in an attempt to increase its expressive power. Thus the increase in expressive power has been at the cost of creeping featurism and the resulting complexity that it brings.

10.1.1.3 Lack of Separation of Synchronisation Code from Sequential Code

One of the parameter types for DRAGOON's behavioural classes is a function that can be invoked by the synchronisation code in order to examine the instance variables of the sequential object being synchronised. This, of course, has the drawback that the programmer of a sequential class may have to write such functions if they do not already exist in the class. Furthermore, as discussed in Section 2.2, access to the instance variables (albeit indirectly via function calls) by synchronisation code is dangerous,

10.1.2 Demeter

Although Lopes and Lieberherr never use the term "generic synchronisation policies," it is clear from the motivation they give in their paper [LL94] that the "synchronisation patterns" of their Demeter language are, in essence, meant to be generic synchronisation policies. Unfortunately, the reality is quite different and synchronisation patterns are far from generic. In particular, the code of a synchronisation pattern is written in terms of the instance variables and actual operations of the class that it is to be instantiated upon. Obviously, this means that a synchronisation pattern can be instantiated upon a different sequential class only if that class happens to contain identically named instance variables and operations. Needless to say, this severely restricts the utility of synchronisation patterns as generic synchronisation policies.

10.1.3 Hector

In HECTOR [BFS93], synchronisation code can be written in a class by itself which will later be instantiated upon a sequential class. However, there are some severe restrictions.

Firstly, it appears that the designers of HECTOR do not realise that synchronisation code can be completely separated from sequential code and thus synchronisation constraints that

examine "state" are specified in a sequential class. Only non-state constraints can be specified in the HECTOR equivalent of a generic synchronisation policy. Thus, the expressive power of generic synchronisation policies is limited.

Secondly, the only formal parameter type of a generic synchronisation policy is an abstract operation name. Note that this is *just* the name; no other information about invocations of operations is available in a generic synchronisation policy. This further limits the expressive power.

Thirdly, a formal parameter of a generic synchronisation policy denotes a single operation, rather than a set of operations. Thus it is impossible to write, say, a generic *ReadersWriter* policy that can be instantiated upon multiple read-style operations and/or multiple write-style operations.

Finally, the technique used to map the formal parameters of a generic synchronisation policy onto actual operations of a class is extremely poor: the first formal parameter is mapped onto the first operation declared in a sequential class, the second formal parameter is mapped onto the second operation declared, and so on.

10.1.4 Parallel Objects

The Parallel Objects (PO) system [CL91] provides parameterised synchronisation constraints. These are similar to generic synchronisation policies in that they are instantiated upon operations. However, their intended usage is different. We have proposed that genericity be applied to entire policies, while PO advocates that genericity be used at the lower level of individual constraints. These parameterised constraints are building blocks which, when instantiated, are combined to form complete policies.

We illustrate PO by an example. The following code uses PO constraints to implement the readers-priority version of the readers/writer policy:

$$\mathit{MaxPar}(\text{Read}) + \mathit{MaxSeq}(\text{Write}) + \mathit{PriQue}(\text{Read, Write})$$

In this code, the *MaxPar* constraint specifies that any number of *Read* invocations can execute together. The *MaxSeq* constraint specifies that a *Write* invocation cannot execute if any other invocation is executing. The final constraint, *PriQue*, specifies that *Read* invocations have priority over *Write* invocations.

A library of predefined constraints is supplied with PO and users can add more if they so wish. Presumably, there is nothing stopping users from writing more complex constraints which are, in fact, complete policies. However, constraints in PO are instantiated upon operations rather than *sets* of operations. Thus it is not clear how one could write, say, a parameterised *ReadersWriter* constraint/policy that could be instantiated upon a set of read-style operations and a set of write-style operations.

Another restriction of the parameterised constraints in PO is the limited expressive power it offers, which is on level with synchronisation counters.

10.1.5 Enabled-sets

One of the main concepts in Enabled-sets is that of *sets of operations*. In this regard, enabled-sets are related to generic synchronisation policies since they too are expressed in terms of sets of operations. However, Enabled-sets are unsuited to denote generic synchronisation policies, as we now discuss.

There is a fundamental difference in how Enabled-sets and GSPs treat sets of operations.

GSPs use sets to denote different types of operations, e.g., *PutOps* and *GetOps* in a bounded buffer, or *ReadOps* and *WriteOps* for a *ReadersWriter* policy. This usage of sets of operations is completely divorced from the details of how a policy might be implemented.

In contrast to this, Enabled-sets associates a set of operations with each *state* of the object being synchronised. This state is, in effect, a synchronisation variable and, as such, an implementation detail of the policy. Such implementation details do not belong in a policy's signature definition.

As an example of how the Enabled-sets concept of *state* is an implementation detail, consider a *ReadersWriter* policy, denoted here in terms of our GSP notation.

policy ReadersWriter[ReadOps, WriteOps]

A typical instantiation of this might be:

ReadersWriter[{Read}, {Write}]

Consider how this policy might be denoted if sets of operations were associated with state. An object might be in one of three states: one or more clients might be *reading* the object, a single client might be *writing* it, or the object might be *idle*. This gives rise to the following definition:

policy ReadersWriter[ReadingStateOps, WritingStateOps, IdleStateOps]

A correct instantiation of this is as follows:

ReadersWriter[{Read}, {}, {Read, Write}]

This can be explained as follows. When in the *reading* state, only other invocations of *Read* may execute. When in the *writing* state, no other operation may execute (because *Write* executes in mutual exclusion)—hence the empty set. Finally, when the object is *idle* both *Read* and *Write* are permitted to execute (though whichever one starts execution first will immediately change the state).

These unintuitive states are implementation details that are of no concern to programmers who simply wish to instantiate the policy. As such, this state information does not belong in the policy's specification.

10.1.6 Summary of Related Work

We have briefly discussed the languages we know of that provide support, in one form or another, for generic synchronisation policies. Although the treatment of genericity varies from one language to another, typical limitations that are common to most of the languages are as follows:

- In most cases, the formal parameters of a policy are instantiated upon individual operations rather than sets of operations.
- The syntax used to instantiate a policy is unintuitive—usually to the point where one does not realise that genericity is being employed.
- It seems that the language designers do not realise that synchronisation code can be fully separated from sequential code without sacrificing expressive power. As such, policies tend to have limited expressive power and/or are not fully generic since they are written in terms of, say, the instance variables of a class.

The support we have provided in GASP for generic synchronisation policies is superior to the support in these other languages and illustrates that it is possible for a language to support generic synchronisation policies without restrictive limitations.

10.2 Summary of Contributions

The question of whether it is feasible to provide language support for generic synchronisation policies has been the focus of Part III of this thesis.

The answer to this question is yes, it *is* feasible to add support for generic synchronisation policies to a language. In Chapter 7 we discussed the concerns that arise in providing language support for GSPs and have resolved them in a satisfactory manner. A summary of these resolutions is as follows:

- As the SOS paradigm shows, it is possible to completely separate synchronisation code and data from sequential code and data without losing any expressive power. As such, the separation from sequential code that genericity imposes upon synchronisation code does not limit the expressive power of a GSP synchronisation mechanism.
- Also as the SOS paradigm shows, in order to have good expressive power, a synchronisation mechanism requires access to just one primary source of information (invocations of operations) rather than a half-dozen or more derivative sources. This means that the formal parameters of a generic synchronisation policy must be of a particular type, which in turn means that this type information need not be stated explicitly. As such, it is not necessary for a host language to be extended to treat "Operation" as a first class type in order to support generic synchronisation policies.
- Generic synchronisation policies can be treated as being akin to classes (even to the point of being able to inherit policies). Being able to model GSPs in an existing language

construct helps minimise the number of extensions to a host language that are required in order to support GSPs.

As a proof of concept that language support for GSPs is feasible, we have added generic synchronisation policies to a host language. As we have shown in Section 10.1, the resulting language, GASP, supports GSPs in a manner that is superior to any other language we are aware of that tries to support GSPs.

Our prototype implementation of GASP, discussed in Chapter 8, shows that generic synchronisation policies can be implemented in a straight-forward manner. The use of GSPs brings with it an execution speed overhead: two extra operation calls per synchronised invocation. However, this is likely to be a small overhead relative to that of synchronisation. If this overhead is felt to be unacceptable then it could be removed by providing, say, compiler support that would in-line the code of a policy's *pre_sync* and *post_sync* into the synchronisation wrapper of a synchronised operation.

While the initial impetus to provide language support for GSPs was to promote code reuse, we have shown that GSPs provide other notable benefits too:

- As discussed in Section 7.3.4, GSPs greatly simplify optimisation by transformation since the compiler knows the name the policy being used rather than having to perform extensive code analysis to deduce this information.
- Adopting a standard object-code interface for generic synchronisation policies permits policies to be written in any synchronisation mechanism. Thus a language could conceivably support several synchronisation mechanisms if it wished to, say, provide a variety of trade-offs between expressive power and efficient implementation. Alternatively, a language might not provide *any* means to write synchronisation policies, thus simplifying the language.
- The instantiation of a generic synchronisation policy is extremely declarative—even if a procedural synchronisation mechanism is used to implement the policy. Thus the use of GSPs can improve the usability of synchronisation mechanisms.

This last point about the declarative nature of instantiating generic synchronisation policies presents a potential irony, as we now discuss.

Papers on DRAGOON claim that, in practice, a relatively small number of behavioural classes (i.e., generic synchronisation policies) will serve most of the synchronisation requirements of programmers [AGMB91, pg. 14] [MCB+89] [Atk90, pg. 111]. (We do not have enough experience with GASP so far to be able to confirm or refute this claim, but its validity seems likely.) Thus, if a library of generic synchronisation policies is shipped with a compiler then most programmers using that compiler may never need to actually write synchronisation code themselves. This suggests that GSPs are more important than the expressive power of synchronisation mechanisms (since a standard library of synchronisation policies need be written only once and users of the library will not be concerned about difficulties

in implementing it). The irony of this is that language support for GSPs is feasible in large part due to the benefits of the SOS paradigm, yet GSPs might make one of these benefits, expressive power, almost redundant.

10.3 Limitations of our Research and Future Work

There are a number of issues regarding generic synchronisation policies that either we have not addressed, or have touched on only briefly. This section briefly summarises these issues.

Firstly, although we have shown that it is possible to have inheritance hierarchies of generic synchronisation policies, we have provided this ability in GASP purely as a means of code reuse. We have not considered subtyping issues for policies. Neither have we considered subtyping issues for classes that instantiate policies.

Secondly, a subclass can either inherit the instantiation of a policy "as is" or re-instantiate it anew. Our prototype implementation does not provide any means for incremental re-instantiation of a policy. Since this issue is of concern to inheritance, we will return to it in Part IV.

A final limitation of our work on language support for GSPs is that we have not addressed the issue of being able to instantiate several policies upon the operations of a class. One of the major stumbling blocks in providing this capability is defining the semantics of combining several policies. This is trivial if the policies to be combined consist of just guards, but unless GSPs are to be restricted to guard-based synchronisation mechanisms, the semantics of combining policies would need to be defined for other synchronisation mechanisms too.

Part IV

Problems with Inheritance in COOPLs

Introduction to Part IV

Inheritance is a mechanism for code reuse that can be found in most object-oriented languages. While inheritance has proved its worth in sequential languages, there are problems with its use in concurrent languages (as we briefly discussed in Section 2.4).

To date, most researchers have thought that the root of the problems lies in a conflict between synchronisation and inheritance. In Chapter 11 we show that the problems are intrinsic to inheritance alone. We discuss two different categories of problems with inheritance and show that the scope of these problems is much wider than previously thought. It is beyond the scope of this thesis to do a detailed analysis of all aspects of the problems with inheritance in COOPLs. Thus, Chapter 11 concentrates its analysis on a subset of the problems.

Chapter 12 contains a survey of how the problems with inheritance in COOPLs manifest themselves in a variety of different inheritance mechanisms. From the results of this survey, we identify the inheritance techniques that help reduce, though not eliminate, the hindrances to reuse associated with the use of inheritance in COOPLs. We also argue that a different reuse mechanism—genericity in the form of generic synchronisation policies—can reduce these hindrances further still.

In the past, several other researchers have tried to analyse the problems associated with the use of inheritance in COOPLs. Unfortunately, most of these researchers have misunderstood the nature of the problems. In Chapter 13, we examine this related work and conclude Part IV of this thesis with a summary of the contributions we have made in this area.

Chapter 11

Analysis of the Problems with Inheritance in COOPLs

In this chapter we discuss two kinds of problem associated with the use of inheritance in COOPLs that can hinder reuse of code.

Snyder [Sny86] has shown that inheritance can violate encapsulation in sequential, object-oriented languages. Section 11.1 shows how similar violations of encapsulation occur in COOPLs.

Another problem with inheritance in COOPLs is that it is employed to reuse two different kinds of code: (i) sequential code, and (ii) synchronisation code. These two uses of inheritance can conflict with each other in a way that hinders the reuse of code. We model this conflict as violations of a particular kind of contract called the *synchronisation policy contract* (SPC). The SPC is introduced in Section 11.2. Then in Section 11.3 we show how inheritance violates it. This leads to a definition of what we term the **I**nheritance of **S**equential code **V**ersus the **I**nheritance of **S**ynchronisation code (ISVIS) conflict in Section 11.4.

A detailed analysis of hindrances to reuse caused by both violations to encapsulation and the ISVIS conflict is beyond the scope of this thesis. Instead, we concentrate on analysing the ISVIS conflict as it applies to the interaction between a common single inheritance mechanism for sequential code and various inheritance mechanisms for synchronisation code. To help with this analysis, we develop the *ISVIS matrix* in Section 11.5. (This matrix will then be employed in the survey of the next chapter.)

Section 11.6 brings this chapter to a close by summarising its main finding.

11.1 Inheritance can Violate Encapsulation

A class exports an interface through which clients can invoke objects of that class. This interface encapsulates (isolates) the implementation details of the class and thus protects clients from possible, future changes to the implementation of the class.

Snyder [Sny86] argues that a subclass is a client of its parent classes and, as such, a

subclass should access its parent class only through an interface—though not necessarily the same interface that is presented to external clients. Without such an interface, a subclass might be written in such a way that is dependent upon the implementation details of a parent class. In other words, inheritance violates encapsulation unless a language provides the ability for a class to define an interface through which subclasses should access it.

Snyder speaks about sequential, object-oriented languages when arguing that inheritance violates encapsulation. However, inheritance can violate encapsulation in COOPLs too. In particular, inheritance can violate the encapsulation of the synchronisation code in a parent class [Mat93] [Ber94]. To see why this is so, consider the following example.

A base class implements two operations, A and B, that examine some instance variables, and a third operation, C, that updates these instance variables. The class also contains some synchronisation code that implements the following policy:

ReadersWriter[{A, B}, {C}]

A subclass introduces a new operation, D, that updates some instance variables and changes the synchronisation code to implement the following policy:

ReadersWriter[{A, B}, {C, D}]

Some time later, a programmer may introduce a new read-style operation, X, to the base class. This necessitates that the synchronisation code of the base class be changed to:

ReadersWriter[{A, B, X}, {C}]

However, it also necessitates that the synchronisation code of the subclass be changed in order to take into account the new operation. Thus we see that the subclass is not encapsulated from changes to the synchronisation code of its parent class.

Partial encapsulation can be achieved by writing the synchronisation code of the subclass as an incremental modification of the synchronisation code in the parent class. For example, the synchronisation code in the subclass might be written as:

super.policy[super.ReadOps, super.WriteOps + {D}]

If, as before, a new read-style operation, X, is introduced to the base class and the synchronisation code of the base class changed to accommodate this new operation then this change will be reflected in the synchronisation code of the subclass.

Similarly, if the generic policy instantiated in the parent class changes from *ReadersWriter* to, say, *ReadersPriority* then this change will also be reflected in the subclass due to its use of "super.policy". However, it should be noted that the only reason this works is because the two policies, *ReadersWriter* and *ReadersPriority*, take identically named formal parameters. If the generic policy instantiated in the parent class changes from *ReadersWriter* to, say, *Alternation*, then the synchronisation code in the subclass will no longer compile since the formal parameters of *Alternation* are *FirstOps* and *SecondOps* rather than *ReadOps* and

WriteOps, as in the *ReadersWriter* policy. Even if the parameters names were identical, the two policies have very different semantics and thus the *Alternation* policy may not be appropriate for use in the subclass.

It is clear that writing the synchronisation code of the subclass as an incremental modification of inherited synchronisation code provides encapsulation against *some* kinds of changes to the synchronisation code of the parent class. We feel the reason that it does not provide *complete* encapsulation is that there is no explicit interface between the parent class and the subclass. Without such an interface, a subclass does not know what aspects of the synchronisation code in its parent class are guaranteed not to change.

We feel it is important a class be able to specify a "synchronisation interface" that is exported to subclasses (perhaps it might be useful to export a synchronisation interface to external clients too). However, it is beyond the scope of this thesis to suggest what form such a language construct might take. As such, we leave this topic for the moment and move on to discuss another problem with the use of inheritance in COOPLS. This other problem, which forms the focus for Part IV of this thesis, is one we call the ISVIS conflict, and we define it in terms of violations of the *synchronisation policy contract.*

11.2 The Synchronisation Policy Contract (SPC)

For the purpose of this discussion, consider a class to be composed of two parts:

- Sequential code. This consists of both the declaration of instance variables/operations, and also the code that implements the operations.
- Synchronisation code. This consists of code (in whatever syntax a synchronisation mechanism is expressed) to implement the synchronisation policy imposed upon objects of a class.

In some languages the sequential code and synchronisation code of a class are syntactically separated. In other languages they are merged. As such, this notion of a class being composed of two separate parts is just a conceptual view and does not always hold at a syntactic level.

A class that contains both sequential code and synchronisation code also (implicitly) contains a *synchronisation policy contract* (SPC). There are three aspects to this contract:

- As its name suggests, the synchronisation policy contract specifies what synchronisation policy is to be used in a class.
- The SPC also specifies some responsibilities for both the synchronisation code and sequential code of the class. The responsibility of the synchronisation code is to correctly implement the specified synchronisation policy.
- The responsibility of the sequential code is that it must implement the operations of the class in such a way as to ensure that, in the face of concurrent invocations constrained by the synchronisation policy, the integrity of the instance variables will be maintained and operations will return correct results.

(Both the sequential code and synchronisation code may have extra responsibilities, in the form of liveness constraints. However, a discussion of this is outside the scope of this thesis.)

As an example, consider a class that contains some instance variables, operations A, B and C, some synchronisation code and the following SPC:[1]

ReadersWriter[{A, B}, {C}]

The synchronisation code's responsibility is to correctly implement this particular policy and the sequential code must ensure that operations A and B do not modify any instance variables. Operation C is free to modify instance variables. In fact, it is *expected* that C will modify instance variables; if this were not so then surely the programmer of the class would have chosen a different SPC.

The next section shows that inheritance may result in the SPC being violated and that such violations can hinder code reuse.

11.3 Inheritance and the SPC

Inheritance is a means of facilitating *reuse*. However, it is also a means of introducing *change* (a subclass that is identical to its parent class would not be of much use). In fact, it can be useful to think of inheritance as a way to obtain code reuse *in spite of* change. We will show in Sections 11.3.1 and 11.3.2 that if change introduced via inheritance violates the SPC of the parent class then it may hinder the reuse of inherited code.

11.3.1 Changes to Inherited Sequential Code and Violations of the SPC

Consider a class that fits the following specification:

Sequential code: instance variables and operations A, B, C
SPC: ReadersWriter[{A, B}, {C}]
Synchronisation code: implements the policy specified in the SPC

Let us suppose that a derived class re-implements operation B and in doing so makes B a write-style operation. The sequential code of the derived class cannot guarantee the integrity of instance variables in the face of concurrent invocations constrained by the inherited synchronisation policy. As such, the programmer must negotiate a new SPC for the derived class. A suitable one would be:

ReadersWriter[{A}, {B, C}]

[1] As previously mentioned, there are three aspects to a synchronisation policy contract: (i) the policy to be used in the class; (ii) responsibilities for the sequential code; and (iii) responsibilities for the synchronisation code. For economy of space we denote a SPC by just the policy alone. The responsibilities of both the sequential and the synchronisation code should be understood implicitly.

Of course, the synchronisation code will have to be changed in order to make it comply with (i.e., implement) this new SPC.

The important point of this example is that a change to the sequential code of a class violated its inherited SPC; this lead to the negotiation of a new SPC which in turn necessitated a change in the synchronisation code (to make it comply with the new SPC).

However, not *all* changes to the sequential code will necessitate changes in the synchronisation code. For example, if, when operation B was re-implemented, it had remained a read-style operation then the SPC would not have been violated and no changes to the synchronisation code would have been required.

Another kind of change to the inherited sequential code of a class that can inhibit reuse of inherited synchronisation code is the introduction of a new operation. For example, consider what will happen if, instead of re-implementing operation B in the derived class, we introduce a new write-style operation, D. In some guard-based mechanisms, invocations on D will be unconstrained by default and thus instance variables will be at risk of corruption. The default in some other mechanisms, e.g., Enabled-sets, is that invocations on D will *never* be serviced; this violates liveness constraints. So, although the effects of adding a new operation varies from one synchronisation mechanism to another, it is clear that doing so violates the SPC and that a new SPC has to be negotiated, e.g.:

ReadersWriter[{A, B}, {C, D}]

As before, the synchronisation code would have to be changed in order to make it comply with the new SPC.

11.3.2 Changes to Inherited Synchronisation Code and Violations of the SPC

It is not only change to the sequential code of a class that might violate the SPC; change to the synchronisation code might also violate the SPC too and inhibit reuse of inherited sequential code.

For example, a programmer may decide to change the synchronisation code of a subclass in order to permit more internal concurrency. Doing so means that the synchronisation part of the class violates the inherited SPC (since it does not implement the inherited synchronisation policy). As a result, the programmer may have to re-implement the sequential operations to ensure the integrity of instance variables in the face of increased concurrency.

Just as not all changes to the sequential code of a subclass will violate the inherited SPC, not all changes to the synchronisation code will violate the inherited SPC either. For example, a synchronisation policy, *Mutex*, might be specified as providing mutual exclusion but without any guarantee on the order that blocked processes will be woken up, other than that the ordering will be *fair*, i.e., it will be impossible for a blocked process to be skipped over indefinitely by other processes. A typical way to ensure fairness is to implement a first-come, first-served scheduling policy, but a programmer might decide to modify the

synchronisation code implementing *Mutex* in order to implement a different, fair scheduling policy, e.g., random order scheduling. Such a change in the synchronisation code of a class would not violate the SPC since the scheduling order was never guaranteed.

11.3.3 Not All Violations Necessitate Change

It should be noted that not all violations of the SPC by sequential code will necessitate change in synchronisation code, and vice versa.

For instance, consider a class that contains a read-style operation, A, and a write-style operation, B, the signatures of which are as follows:

A(Size: Int, ...)
B(Size: Int, ...)

The sequential code of this class is compatible with several synchronisation policies, including:

ReadersWriter[{A}, {B}]
Mutex[{A, B}]
SJN[{A, B}, Size]

Let us suppose that the SPC chosen for this class is:

ReadersWriter[{A} {B}]

and that the synchronisation code implements this policy. Consider a derived class that changes the synchronisation code so that it implements one of the other listed policies, e.g.:

SJN[{A, B}, Size]

This obviously violates the SPC, yet it does not necessitate any changes in the sequential code (since the sequential code is already compatible with the new policy).

Just as the sequential code of a class might be compatible with several synchronisation policies, similarly it is possible for a class's synchronisation code to comply with, i.e., implement, several synchronisation policies.

One example has already been given: code that implements a first-come, first-served scheduler also implements a mutual exclusion policy. Also, if a mutual exclusion policy does not guarantee fairness then a *SJN* policy will be compatible with it.

Another example can be found in the Eiffel|| language. Code implementing the first-come, first-served policy is written in a manner that is independent of the names, and number, of operations upon which it is being instantiated. Thus, the same code can implement, say, "FCFS[{A, B, C}]" for a base class and: "FCFS[{A, B, C, D}]" in a derived class that introduces a new operation, D. However, it should be noted that this applies to Eiffel||'s implementation of first-come, first-served *only*; in general, instantiating a synchronisation policy on an increased number of operations in Eiffel|| requires that the synchronisation code be completely rewritten.

11.4 Summary of the SPC and Definition of the ISVIS Conflict

A class that contains both sequential code and synchronisation code also (implicitly) contains a *synchronisation policy contract* (SPC). The SPC specifies what synchronisation policy is to be used in a class and some responsibilities for both the sequential code and the synchronisation code of the class. When change is introduced to the sequential code (or synchronisation code) of a class, perhaps via inheritance, it is possible that the SPC will be violated. This will necessitate a re-negotiation of the SPC and the synchronisation code (or sequential code) may have to be changed in order to make it comply with the new SPC. Thus, when a class contains both sequential code and synchronisation code, changes to one in a subclass may necessitate change in (and hence hinder the reuse of) the other. This is the **I**nheritance of **S**ynchronisation code **V**s. the **I**nheritance of **S**equential code (ISVIS) conflict.

It is instructive to draw an analogy between solving the ISVIS conflict and curing medical conditions. Some medical conditions that afflict humans cannot be *cured* but can be *controlled* to reduce their harmful effects. For example, poor eyesight is usually not curable (except perhaps by expensive surgery) but can be controlled by the prescription of spectacles or contact lens. Similarly, there is no cure for diabetes but it can be controlled via insulin. The ISVIS conflict is similar in that our analysis suggests that it is *intrinsic* to languages that contain both sequential code and synchronisation code and hence the conflict cannot be "cured" (solved). However, this does not preclude the possibility of *controlling* the conflict to reduce its harmful effects.

This distinction between *solving* the conflict and *controlling its harmful effects* is one that is rarely made and numerous researchers claim to have solved the ISVIS conflict when it would be more accurate for them to claim that they have developed techniques to control its harmful effects [Löh93] [MY90] [GW91] [Neu91] [BAWY92] [Tho94] [KL89] [BI92] [Mes93] [BFS93] [LL94].

11.4.1 A Common Misunderstanding of the Problems with Inheritance in COOPLs

Numerous researchers have referred to hindrances to reuse in COOPLs as a "conflict between synchronisation and inheritance" (or words to similar effect) [TS89] [KL89] [Neu91] [Tho94] [BBI+] [Mes93] [BFS93] [Mat93] [Ber94]. This phrasing suggests that it might be possible to solve the conflict by developing new synchronisation mechanisms that do *not* interfere with inheritance. Our analysis so far of the problems shows that this perception is incorrect and that the problems are, in fact, intrinsic to inheritance:

- The ISVIS conflict is due to two different uses of inheritance conflicting with one another.
- Inheritance can also violate encapsulation, thus hindering code reuse.

The fact that the problems are intrinsic to inheritance itself rather than being a conflict between synchronisation and inheritance has several ramifications.

Firstly, approaches to tackling the problems need to be focussed on, say, designing new inheritance models or alternative ways to reuse code, rather than on designing new synchronisation mechanisms.

Secondly, the problems may not be confined to the niche of synchronisation, but may show up in other areas where inheritance is employed as a means to reuse several different types of code in a single class. For example, similar problems have been noted in the area of real-time constraints [ABvdSB94].

11.5 Analysing the ISVIS Conflict

A lot of past research into the ISVIS conflict has focussed on just one or two programming exercises involving inheritance that display the conflict. Some researchers have designed mechanisms for inheriting synchronisation code that implement these programming exercises gracefully and thus concluded that their mechanisms tackle the conflict [KL89] [TS89] [Cou94] Such claims are invariably flawed. The only thing these researchers have shown is that their mechanisms tackle one or two particular forms that the conflict might take. This does not mean that their mechanisms cope equally well with the conflict in *other* forms.

Using just one or two programming exercises to test how an inheritance mechanism copes with the ISVIS conflict is clearly not sufficient. One might employ a larger number of programming exercises [Mat93] [BI92] [Mes93] [Löh93] [LL94] [Ber94] [Tho94] but still one would not have any assurance that they embody *all* the forms that the conflict might take. A more systematic approach is needed.

It seems obvious that the way to approach this is to (somehow) enumerate all the possible circumstances in which the conflict might arise. Having done this, one could then devise a set of programming exercises that test all of these different circumstances and use these exercises to evaluate how different inheritance mechanisms tackle the conflict. In this way one can be assured that a set of programming exercises is comprehensive enough to test for all the ways that the conflict might manifest itself.

The key here is to enumerate all the possible circumstances in which the conflict might arise. Towards this goal, we now introduce the *ISVIS matrix*.

11.5.1 The ISVIS Matrix

The ISVIS conflict arises due to a subclass changing either sequential code or synchronisation code (or both). As such, we start by enumerating the different ways in which a subclass might change code.

Having inherited from a base class, the following possible changes may take place to the sequential code in the derived class:

- It might not change at all.
- It might change in a way that does not violate the inherited SPC.
- It might change in a way that *does* violate the inherited SPC.

Similar changes may take place in the synchronisation code of a derived class.

This information could be combined to form a matrix like that shown in Figure 11.1. This matrix enumerates the possible interactions between the changes that can occur in both the sequential code and synchronisation code. However, a few useful modifications can be made to it.

sync code / seq code	no change	changes that do not violate the SPC	changes that do violate the SPC
no change			
changes that do not violate the SPC			
changes that do violate the SPC			

Figure 11.1: Possible combinations of changes that can occur in a derived class

First of all, we can split up the third row ("changes that violate the SPC") into two parts:

- Changes that violate the SPC but do not necessitate change of synchronisation code.
- Changes that violates the SPC and necessitate change of synchronisation code.

The first of these will be merged with the second row and that row renamed "changes that do not necessitate change in synchronisation code." The other will then form the third row by itself. Similar changes can be made to the columns of the matrix.

The resulting matrix is shown in Figure 11.2.

The final step of refining the ISVIS matrix into a useful tool is dependent upon the inheritance model used for sequential code. There are many variations of inheritance for sequential code. For example:

- Some languages support a basic form of single inheritance in which a subclass can inherit an operation "as is," re-implement it anew or re-implement it but include a "super" call to invoke the operation in the parent class.

seq code \ sync code	no change	changes that do not necessitate change in seq code	changes that necessitate change in seq code
no change	(1, 1)	(1, 2)	(1, 3)
changes that do not necessitate change in sync code	(2, 1)	(2, 2)	(2, 3)
changes that necessitate change in sync code	(3, 1)	(3, 2)	(3, 3)

Figure 11.2: The ISVIS Matrix

- Some languages, e.g., Eiffel [Mey92], permit a subclass to "undefine" an inherited operation.
- Some languages support multiple inheritance. Actually, there are many different forms of multiple inheritance since different languages have, say, different means of resolving naming conflicts and different ways of handling repeated inheritance.
- The Beta language [KMMPN87] provides an "inner" construct which is, in effect, the opposite of "super." Rather than have an operation in a subclass make a "super" call to call *up* the inheritance hierarchy, an operation in a parent class makes an "inner" call to call *down* the inheritance hierarchy.
- Bracha and Cook propose "mixin-based inheritance" [BC90] which, they claim, is a generalisation of Beta-style inheritance, single inheritance with "super" calls and multiple inheritance.
- Some languages provide delegation as an alternative to inheritance (though there are some claims that delegation is actually a form of inheritance [Ste87]). This is another area that needs to be analysed for conflicts that might hinder code reuse [BY87].

It is beyond the scope of this thesis to analyse how the ISVIS conflict manifests itself in all of these different forms of inheritance for sequential code. Instead, we confine ourselves to analysing how the ISVIS conflict manifests itself in the first form of inheritance given in the list, i.e., single inheritance in which a subclass might change the inherited sequential code as follows:[2]

[2] Note that possible changes to instance variables in a subclass are of no concern to this discussion since they will not be synchronised directly, but only via operations that access them.

(a) An operation may be re-implemented anew. We term this a "total re-implementation" of the operation.

(b) An operation may be re-implemented, but the new code may include a "super" call to the operation in the parent class. We refer to this as an "incremental modification" of the operation.

(c) A new operation may be introduced.

The re-implementation of an operation—whether it takes the form of (a) or (b)—may or may not necessitate a change in synchronisation code. For example, when inheriting from a class that employs a *ReadersWriter* policy, the re-implementation of a read-style operation will not necessitate a change in synchronisation code if the re-implemented operation remains a read-style operation; but will necessitate a change in synchronisation code if the re-implemented operation becomes a write-style operation. As such, changes (a) and (b) belong in both the second and third rows of the matrix. The introduction of a new operation always necessitates change to the synchronisation code[3] so change (c) belongs only in the third row of the matrix. The reason for this subdivision of the rows is that it is possible that the conflict might manifest itself differently for each of the three kinds of change—(a), (b) and (c). The resulting matrix is shown in Figure 11.3.

sync code / seq code	no change	changes that do not necessitate change in seq code	changes that necessitate change in seq code
no change	(1, 1)	(1, 2)	(1, 3)
changes that do not necessitate change in sync code	(a)		
	(b) (2, 1)	(2, 2)	(2, 3)
changes that necessitate change in sync code	(a)		
	(b)		
	(c) (3, 1)	(3, 2)	(3, 3)

Figure 11.3: The ISVIS Matrix for single inheritance (and concurrency within objects)

We do not attempt to subdivide columns two and three of the matrix in the way that we have subdivided rows two and three. This is because mechanisms for inheriting synchronisation code are relatively new and less well understood than mechanisms for inheriting

[3] Actually, *nearly* always. As discussed in Section 11.3.3, one exception to this is the Eiffel|| implementation of FCFS which does not need to be changed in order to accommodate new operations.

sequential code. As such, it is difficult to know how the columns might be subdivided meaningfully.

11.5.2 Using the ISVIS Matrix to Develop Sets of Programming Exercises

Now that the ISVIS matrix enumerates the different ways in which the ISVIS conflict might occur for the single-inheritance (of sequential code) model we have chosen to analyse, one could, in principle, devise a set of programming exercises that test each cell of the matrix. This set of exercises could then be used to evaluate how the ISVIS conflict manifests itself when this inheritance (of sequential code) mechanism is combined with various inheritance (of synchronisation code) mechanisms.

Ideally, we would use just one synchronisation mechanism when performing this evaluation. After all, we wish to evaluate different inheritance mechanisms rather than different synchronisation mechanisms. However, when an inheritance (of synchronisation code) mechanism is developed, it is usually developed for a particular synchronisation mechanism. In some cases, the inheritance mechanism and the synchronisation mechanism are coupled together tightly and are not easily separated. Thus, in order to evaluate a variety of inheritance (of synchronisation code) mechanisms, we are forced to employ a variety of synchronisation mechanisms. This raises a potential problem, as we now discuss.

Some synchronisation mechanisms generally keep synchronisation code separated from sequential code but resort to mixing the two together in order to implement a synchronisation policy that is beyond the expressive power of the synchronisation mechanism. This mixing of code can potentially hinder inheritance. As such, we must ensure that none of the programming exercises used in the evaluation of an inheritance (of synchronisation code) mechanism require expressive power beyond that available in the synchronisation mechanism. If we do not ensure this then a programming exercise requiring more expressive power might induce the ISVIS conflict and thus prejudice the evaluation of an inheritance mechanism. Thus, in order to avoid this possibility, we must choose programming exercises that do not over-stretch the expressive power of synchronisation mechanisms. This, unfortunately, precludes the possibility of having a single programming exercise per cell that could be used in the evaluation of all inheritance (of synchronisation code) mechanisms. There are several reasons for this.

Firstly, for a set of programming exercises to be usable for the evaluation of *all* inheritance (of synchronisation code) mechanisms would mean that the expressive power required to implement any of the policies used in the exercises would have to be no greater than the lowest common denominator of expressive power available in all synchronisation mechanisms. This lowest common denominator of expressive power would be so poor as to make it quite difficult to develop a complete set of programming exercises.

Also, many synchronisation mechanisms contain several constructs, e.g., Esp contains guards and actions. It is possible that each of the constructs in a synchronisation mechanism is inherited in a different way. Thus to evaluate how a particular inheritance mechanism copes with the ISVIS conflict, it is necessary to evaluate it with each of the constructs in a

synchronisation mechanism. A set of programming exercises based on a low level of expressive power might not exercise all constructs.

Finally, one might hope that a universal set of programming exercises would contain exactly one exercise per cell of the ISVIS matrix. While this might seem like a good starting point, it is possible that an inheritance mechanism for synchronisation code might have only *partial* support for a particular cell of the matrix, and that it would gracefully handle one programming exercise within this cell while not fairing so well on a different exercise in that same cell. (Examples of this will be seen in the next chapter.)

For the above reasons, we will develop *several* sets of exercises on an as-needed basis. In this way, we can ensure that the evaluation of an inheritance (of synchronisation code) mechanism is not prejudiced by it being associated with a synchronisation mechanism of poor expressive power.

11.5.3 Two ISVIS Matrices

The ISVIS matrix shown previously (Figure 11.3) is suitable for languages that permit concurrency within objects. However, some parts of this matrix do not apply to languages that do not support concurrency within an object.[4]

For example, the third column of the matrix is for changes to synchronisation code that necessitate changes in sequential code. The only kind of change that we can envisage applying to this column is to change the synchronisation code in order to permit more (or less) internal concurrency, e.g,. from a *Mutex* policy to a *ReadersWriter* policy. Of course, one can only have *more* internal concurrency in an object if the host language permits internal concurrency in the first place. Thus it is clear that this column would not make sense in a language that does not permit concurrency within objects.

Similarly, rows 3a and 3b only apply to languages that allow internal concurrency. This is because the only way that an incremental or total re-implementation of an existing operation will invalidate the SPC of a class is if the language permits internal concurrency and the re-implementation changes an operation from being, say, a read-style operation into a write-style operation.

Thus, rows 3a and 3b, and column 3 of the ISVIS matrix are disregarded when evaluating inheritance (of synchronisation code) mechanisms in languages that do not support concurrency within objects. The resulting matrix is shown in Figure 11.4.

11.6 Summary

It is well known that there are problems associated with the use of inheritance in COOPLs. A common perception is that the problems are rooted in a conflict between synchronisation

[4] Note that a language that does not support concurrency *within* an object is still considered to be a concurrent language if it supports concurrency *between* objects. In this case, the synchronisation code inside an object will be concerned solely with the scheduling of pending invocations.

seq code \ sync code	no change	changes that do not necessitate change in seq code
no change	(1, 1)	(1, 2)
changes that do not necessitate change in sync code	(a) / (b) (2, 1)	(2, 2)
changes that necessitate change in sync code	(c) (3, 1)	(3, 2)

Figure 11.4: The ISVIS Matrix for single inheritance (and no concurrency within objects)

and inheritance. In this chapter we have shown that this perception is incorrect and that the problems are, in fact, intrinsic to inheritance.

We have divided the problems into categories. One is that inheritance violates encapsulation; the other, which we call the ISVIS conflict, is that trying to inherit two different types of code can result in hindrances to code reuse. A detailed study of both categories of problem is outside the scope of this thesis. Instead, we have confined ourselves to analysing the ISVIS conflict in the combination of a common single-inheritance mechanism for sequential code with inheritance mechanisms for synchronisation code. We have developed a tool, the ISVIS matrix, that allows us to develop comprehensive sets of programming exercises to see how the ISVIS conflict manifests itself in these inheritance mechanisms. The next chapter employs the ISVIS matrix in a survey of different inheritance mechanisms for synchronisation code.

Chapter 12

Reducing the ISVIS Conflict's Harmful Effects: A Survey and Proposal

The previous chapter defined the ISVIS conflict and developed the ISVIS matrix. In this chapter we employ the ISVIS matrix to survey the conflicts that arise when various inheritance mechanisms used for synchronisation code interact with the inheritance of sequential code.

The results of the survey will support our analysis in the previous chapter that the problem with the use of inheritance in COOPLs is intrinsic to inheritance rather than being due to existing synchronisation mechanisms.

We conclude the chapter by showing how an alternative reuse mechanism, genericity (in the form of generic synchronisation policies) can substantially reduce the harmful effects of the ISVIS conflict.

Readers may find it useful to photocopy the ISVIS matrices (Figures 11.3 and 11.4) introduced in the last chapter and keep these photocopies to hand when reading the survey.

12.1 Inheritance of Synchronisation Counters

Numerous guard-based mechanisms employ synchronisation counters. However, each mechanism usually has its own extensions/idiosyncrasies so we we start off by evaluating the inheritance of a hypothetical, guard-based synchronisation mechanism that can access *just* synchronisation counters. (Other guard-based mechanisms will be examined later.) We assume that this mechanism supports the modification of guards by, say, an **inherits** keyword, similar to that proposed for Guide [DDR+91, RR92].

Before we start the analysis, we should point out that there is a lot of confusion in the literature over inheritance of guards in Guide. There are two reasons for this.

Firstly, an early Guide paper [DKM+88] contained an ambiguously worded paragraph which could be interpreted to mean that that if a subclass changes the guard of an operation

then the sequential code of that operation must be re-implemented too. In fact, guards and the sequential code of operations are kept separate and a re-implementation of one in a subclass need not necessitate a re-implementation of the other. Thankfully, later papers on Guide [DDR+91] [Riv92] [RR92] are clearer on this point.

A second source of confusion regarding Guide is its ability to incrementally modify guards. By default, a subclass inherits all guards "as is." However, a subclass can incrementally modify guards by use of the **inherits** keyword. For example, a guard might be of the form:

Foo: **inherits and** ...;

This states that the guard for operation *Foo* is similar to the guard in the parent class but incrementally modified by **and**-ing on additional constraints. In effect, the **inherits** keyword is akin to a "super" call in an operation. (Of course, instead of incrementally modifying a guard, a subclass can re-implement it anew, if it so wishes.) The ability to incrementally modify guards has not always been present in Guide, and it is only recent papers that discuss it [DDR+91] [Riv92] [RR92]. As such, many researchers are apparently unaware that Guide has this capability.

Because of these two sources of confusion, do not be surprised if the survey of inheritance for guard-based mechanisms in this chapter shows them to be better than is indicated by previous reviews of Guide in the literature [Löh93] [Mat93] [Ber94].

12.1.1 1st Column of the Matrix

We begin with programming exercises that test the cells in the first column of the ISVIS matrix. Note that cell(1, 1) of the matrix does not involve any change at all. As such, we ignore it and start off with a programming exercise to illustrate cell(2, 1).

1st Programming Exercise—cell(2a, 1)) and cell(2b, 1)

Write a synchronised class that contains operations A, B and C, and has synchronisation code that implements the following SPC:

ReadersWriter[{A, B}, {C}]

Inherit from this class and do the following:

- Totally re-implement A, keeping it as a read-style operation. (This tests cell(2a, 1)).
- Incrementally modify B, keeping it as a read-style operation. (This tests cell(2b, 1)).

The base and derived classes to implement this exercise are shown in Figure 12.1. In this case we can see that, as expected, the modification of the sequential code did not hinder the reuse of the inherited synchronisation code.

```
class Base {
   A(...) { ... }   B(...) { ... }
   C(...) { ... }
synchronisation
   A, B: exec(C) = 0;
   C: exec(A, B, C) = 0;
}
```

```
class Derived inherits Base {
   A(...) { ... }
   B(...) { super.B(...); ... }
   // no change to C
synchronisation
   // no change to sync code
}
```

Figure 12.1: 1st Programming exercise (synchronisation counters)

2nd Programming Exercise—cell(3c, 1)

Using the same base class as in the previous programming exercise, inherit and make the following change to the derived class:

Introduce a new write-style operation, *D*.

Introducing this new operation violates the inherited SPC and so the following new one is adopted:

ReadersWriter[{A, B}, {C, D}]

Of course, the synchronisation code will have to be modified to make it conform to this new SPC. The code for this exercise is shown in Figure 12.2. Note that because of the similarity between the synchronisation policies in the base and derived classes (both are instantiated from the same generic policy), we are able to change from one to the other by means of **and**-ing on new constraints to the inherited guards.

```
class Base {
   A(...) { ... }   B(...) { ... }
   C(...) { ... }
synchronisation
   A, B: exec(C) = 0;
   C: exec(A, B, C) = 0;
}
```

```
class Derived inherits Base {
   // no change to A, B, C
   D(...) { ... }
synchronisation
   A, B, C: inherits and exec(D) = 0
   D: exec(A, B, C, D) = 0
}
```

Figure 12.2: 2nd Programming exercise (synchronisation counters)

However, it is worth noting that it would have been just as short (in terms of number of keystrokes), if not slightly shorter, to implement the synchronisation policy in the derived class anew rather than incrementally modify the inherited one. Furthermore, it is more difficult to write and understand the guards in the derived class if they incrementally modify inherited guards rather than are written anew, as shown below:

A, B: *exec*(C, D) = 0;
C, D: *exec*(A, B, C, D) = 0;

This disadvantage of incrementally modifying inherited guards is not unique to this programming exercise; it occurs in many of the other programming exercises too.

3rd Programming Exercise—cell(3a, 1) and cell(3b, 1)

Using the same base class as in the previous programming exercises, inherit and make the following change to a derived class:

> Totally re-implement B in a manner that makes it become a write-style operation.

Making this change violates the inherited SPC and so the following new one is adopted:

> ReadersWriter[{A}, {B, C}]

Of course, the synchronisation code will have to be modified to make it conform to this new SPC. The code for this exercise is shown in Figure 12.3. Once again, because of the similarity between the synchronisation policies in the base and derived classes, we are able to change from one to the other by means of **and**-ing on new constraints to the guards. Also, again, it would have been slightly shorter and easier to write and understand the synchronisation code if it were written anew (as shown below) rather than incrementally modified:

> A: *exec*(B, C) = 0;
> B, C: *exec*(A, B, C) = 0;

This is not to suggest that programmers *should* write synchronisation policies anew rather than try to reuse existing code. On the contrary, we claim that this reuse mechanism is inadequate and a better one needs to be developed. Further proof of this claim will come through additional programming exercises, both for this synchronisation mechanism and also for other guard-based mechanisms discussed later.

A final note: this programming exercise involved a total re-implementation of B and hence tested cell(3a, 1); however, the reader should be easily able to verify that if B had been incrementally modified (thus testing cell(3b, 1)) then the same results would have been obtained.

```
class Base {
  A(...) { ... }  B(...) { ... }
  C(...) { ... }
synchronisation
  A, B: exec(C) = 0;
  C: exec(A, B, C) = 0;
}
```

```
class Derived inherits Base {
  B(...) { ... }
synchronisation
  A: inherits and exec(B) = 0
  B: inherits and exec(A, B) = 0
}
```

Figure 12.3: 3rd Programming exercise (synchronisation counters)

4th Programming Exercise—cell(3a, 1) and cell(3b, 1)

This exercise is similar to the previous one, except that instead of changing B from a read-style operation to a write-style operation, we go the other way, i.e., change B from a write-style operation to a read-style operation. In other words, the SPC of the base class is:

ReadersWriter[{A}, {B, C}]

In the derived class, operation B will be re-implemented and the SPC changed to be:

ReadersWriter[{A, B}, {C}]

As in the previous programming exercise, this tests cell(3a, 1) and can be easily adapted so that it tests cell(3b, 1). The code for this exercise is shown in Figure 12.4. In this case, there is a similarity between the synchronisation policies of the base and derived classes but we want to *remove* constraints rather than *add* them in order to go from one policy to the other. Since no way is provided to achieve this form of incremental modification, the guards for operations A and B must be rewritten anew.

```
class Base {
  A(...) { ... }  B(...) { ... }
  C(...) { ... }
synchronisation
  A: exec(B, C) = 0;
  B, C: exec(A, B, C) = 0;
}
```

```
class Derived inherits Base {
  B(...) { ... }
synchronisation
  A, B: exec(C) = 0;
  // guard for C can be reused
}
```

Figure 12.4: 4th Programming exercise (synchronisation counters)

The facilities for incrementally modifying guards have failed in this programming exercise. They also prove inadequate in the next.

5th Programming Exercise—cell(3c, 1)

Write a synchronised bounded buffer class that contains operations *Put* and *Get*, and has the following SPC:

BBuf[{Put}, {Get}, Size]

Inherit and making the following change in the derived class:

Introduce a new operation, *PutFront*.[1]

Making this change violates the inherited SPC and so the following new one is adopted:

[1] As its name suggests, *PutFront* inserts an item at the front of the buffer (while *Put* inserts an item at the rear of the buffer). If clients forgo the use of *Put* and instead access the buffer only via *PutFront* and *Get* then the buffer is, in effect, used as a stack.

BBuf[{Put, PutFront}, {Get}, Size]

Of course, this necessitates a change in the synchronisation code so that it will comply with this new SPC. The code for the base class is shown in Figure 12.5.

```
class Buffer[Size: Int] {
  Put(...) { ... }
  Get(...) { ... }
synchronisation
  Put: exec(Put, Get) = 0 and term(Put) − term(get) < Size;
  Get: exec(Put, Get) = 0 and term(Put) − term(get) > 0;
}
```

Figure 12.5: 5th Programming exercise (synchronisation counters)

Notice that the calculation of the number of items currently in the buffer is given by the expression:

$$term(\text{Put}) - term(\text{Get})$$

When the derived class introduces a new operation, *PutFront*, this expression must be replaced with:

$$term(\text{Put, PutFront}) - term(\text{Get})$$

This cannot be achieved by, say, **and**-ing a new constraint onto the inherited guards and so we are unable to reuse *any* of the guards, but instead have to write them anew.

This example of adding *PutFront* to a buffer is frequently used in the literature. Another common example from the literature which also tests cell(3c, 1) is for the subclass to add a new operation, *Get2*, that removes the first two items from the buffer.[2] If the *Get2* exercise were used then the results would be the same: the subclass would have to write the guards anew.

12.1.2 2nd Column of the Matrix

The programming exercises of the first column of the ISVIS matrix centered around the sequential code of a class changing and how this affected synchronisation code. Now we move onto the second column in which the synchronisation code is changed by having the derived class instantiate a different generic policy. However, these changes are such that they should not necessitate changes to the sequential code.

[2] An operation such as *Get2* might be defined if it was necessary to ensure that two items obtained from the buffer were adjoining items. This could not be guaranteed by two calls to *Get* due to the possibility of concurrent invocations of *Get* by other clients.

6th Programming Exercise—cell(1, 2)

Write a synchronised base class that contains operations A, B and C, and the following SPC:

ReadersWriter[{A, B}, {C}]

Inherit from this class and in the derived class change the SPC to be:

ReadersPriority[{A, B}, {C}]

The code for this exercise is shown in Figure 12.6. As can be seen, due to the similarity between the two generic synchronisation policies, it is possible to express the guards in the derived class as a incremental modification of the guards in the base class. Also, in contrast to a re-instantiation of the same generic policy, the reusing/modifying of the guard on operation C is relatively easy to write and understand.

```
class Base {
  A(...) { ... }  B(...) { ... }
  C(...) { ... }
synchronisation
  A, B: exec(C) = 0;
  C: exec(A, B, C) = 0;
}
```

```
class Derived inherits Base {
  // no change to seq code
synchronisation
  C: inherits and wait(A, B) = 0
}
```

Figure 12.6: 6th Programming exercise (synchronisation counters)

However, if the base class had employed the *ReadersPriority* policy and the derived class the more basic *ReadersWriter* policy then we would have had wanted to *remove* the extra constraint in order to go from one policy to the other. Since there is no way to achieve this form of modification, the guard for operation C would have had to be rewritten anew.

The remaining programming exercises exhaustively test the other cells of the matrix. However they do not reveal much more of interest. As such, readers may wish to skip ahead to Section 12.1.4, where we summarise what the ISVIS matrix has told us about this synchronisation mechanism.

7th Programming Exercise—cell(2, 2)

This exercise is similar to the previous one, except that the derived class makes some changes to the sequential code as well as to the synchronisation code. The changes to the sequential code do not violate the inherited SPC and hence should not necessitate changes to the synchronisation code.

Write a synchronised base class that contains operations A, B and C, and the following SPC:

ReadersWriter[{A, B}, {C}]

Inherit from this class and in the derived class make the following changes:

- Change the synchronisation code so that it implements:

  ReadersPriority[{A, B} {C}]

- Totally re-implement A, keeping it as a read-style operation. (This tests cell(2a, 2)).
- Incrementally modify B, keeping it as a read-style operation. (This tests cell(2b, 2)).

There is little point in showing the code for this exercise since it is similar to that of the previous exercise (the code of which is shown in Figure 12.6), except that operations A and B are modified as mentioned above. This modification of the sequential operations does not hinder the inheritance of synchronisation code at all.

8th Programming Exercise—cell(3, 2)

As in the two previous examples, the derived class instantiates a generic policy different to that used in the base class. This change in policy should not affect the sequential code. However, the programmer makes some changes to the sequential code anyway—changes that would have necessitated change in the synchronisation code even if a programmer had not planned on changing the synchronisation code.

Write a synchronised base class that contains operations A, B and C, and the following SPC:

ReadersWriter[{A, B}, {C}]

Inherit from this class and in the derived class make the following changes:

- Change the synchronisation code so that it implements a *ReadersPriority* policy.
- Totally re-implement C, and in doing so change it into a read-style operation. (This tests cell(3a, 2)).
- Incrementally modify A, and in doing so change it into a write-style operation. (This tests cell(3b, 2)).
- Introduce a new write-style operation, D. (This tests cell(3c, 2)).

The code for this exercise is shown in Figure 12.7. As expected, the change from the basic *ReadersWriter* policy to *ReadersPriority* does not have any effect on the sequential code (although, as in the 6th programming exercise, it would have if the change in the policy had gone the other way round). However, the changes made to the sequential code *do* affect the synchronisation code and the end result is that the synchronisation code is written anew. If the sequential had not changed in this manner then the change in the synchronisation code might have been expressed as an incremental modification of the inherited synchronisation code (as in the previous two exercises).

```
class Base {
  A(...) { ... }
  B(...) { ... }
  C(...) { ... }
synchronisation
  A, B: exec(C) = 0;
  C: exec(A, B, C) = 0;
}
```

```
class Derived inherits Base {
  A(...) { super.A(...); ... }
  C(...) { ... }
  D(...) { ... }
synchronisation
  B, C: exec(A, D) = 0;
  A, D: exec(A, B, C, D) = 0
                    and wait(B, C) = 0;
}
```

Figure 12.7: 8th Programming exercise (synchronisation counters)

12.1.3 3rd Column of the Matrix

The remaining programming exercises test the third column of the ISVISmatrix, i.e., the derived class will change the synchronisation code in such a manner that requires the sequential code to be changed. The only type of change to synchronisation code that can have this effect is a change that permits more internal concurrency. This is generally achieved by relaxing, i.e., removing, some constraints on operations. Since the means to incrementally modify a guard do not permit existing constraints to be removed, we can expect that most guards that change will have to be rewritten anew.

9th Programming Exercise—cell(1, 3)

Write a synchronised base class that contains operations A, B and C, and the following SPC:

ReadersWriter[{A}, {B, C}]

Inherit from this class and change the synchronisation code so that it implements:

ReadersWriter[{A, B}, {C}]

This policy permits more internal concurrency. Of course, it necessitates a change to the sequential code: operation B will have to be re-implemented in order to make it into a read-style operation. The code for this exercise is shown in Figure 12.8. As expected, the two guards affected by the change in the synchronisation policy, A and B, had to be rewritten anew. The guard for operation C was not affected and hence could be inherited and reused without change.

10th Programming Exercise—cell(2, 3)

As in the previous exercise, write a synchronised base class that contains operations A, B and C, and the following SPC:

ReadersWriter[{A}, {B, C}]

```
class Base {
   A(...) { ... }   B(...) { ... }
   C(...) { ... }
synchronisation
   A: exec(B, C) = 0;
   B, C: exec(A, B, C) = 0;
}
```

```
class Derived inherits Base {
   B(...) { ... }
synchronisation
   A, B: exec(C) = 0;
   // no change to guard for C
}
```

Figure 12.8: 9th Programming exercise (synchronisation counters)

Inherit from this class and change the synchronisation code so that it implements:

ReadersWriter[{A, B}, {C}]

As before, this necessitates that operation B be re-implemented in order to make it into a read-style operation. Make the following additional changes to the sequential code:

- Totally re-implement A in a manner that keeps it as a read-style operation. (This tests cell(2a, 3).)
- Incrementally modify C in a manner that keeps it as a write-style operation. (This tests cell(2b, 3).)

The code for is exercise is similar to that of the previous exercise (shown in Figure 12.8). The only difference is that the derived class re-implements operations A and C. These additional changes to the sequential code do not affect the synchronisation code.

11th Programming Exercise—cell(3, 3)

As in the previous exercise, write a synchronised base class that contains operations A, B and C, and the following SPC:

ReadersWriter[{A}, {B, C}]

Inherit from this class and make the following changes:

- Change the synchronisation code so that it treats B as a read-style operation. As in the previous two exercises, this necessitates that operation B be re-implemented.
- Totally re-implement A, in a way that changes it into a write-style operation. (This tests cell(3a, 3)).
- Introduce a new read-style operation, D. (This tests cell(3c, 3)).

The resulting SPC of the subclass is:

ReadersWriter[{B, D}, {A, C}]

The code for this exercise is shown in Figure 12.9. As one might expect, since changes are made to both the sequential code and the synchronisation code, there is not much reuse. This exercise does not test cell(3b, 3). This can be remedied by changing the exercise so that operation A is incrementally modified instead of being re-implemented anew.

```
class Base {
  A(...) { ... }  B(...) { ... }
  C(...) { ... }
synchronisation
  A: exec(B, C) = 0;
  B, C: exec(A, B, C) = 0;
}
```

```
class Derived inherits Base {
  A(...) { ... }
  B(...) { ... }
  D(...) { ... }
synchronisation
  B, D: exec(A, C) = 0;
  A: exec(A, B, C, D) = 0;
  C: inherits and exec(D) = 0;
}
```

Figure 12.9: 11th Programming exercise (synchronisation counters)

12.1.4 Discussion

We now summarise what has been learned about the inheritance of guards that contain just synchronisation counters.

Guards can be inherited "as is," totally rewritten or incrementally modified by combining them with new constraints via the **and** boolean operator.[3] While this incremental modification of guards is a technique for reuse, it is not an ideal one, for several reasons:

- It is as verbose, if not slightly more so, than simply rewriting guards anew.
- It is also more difficult to write guards this way, and more difficult to understand them later.
- Finally, it can only be used to add new constraints to guards. It cannot be used to replace or remove existing constraints.

12.2 Inheritance of Scheduling Predicates

Incremental modification of inherited guards that contain scheduling predicates is similar to the incremental modification of inherited guards that contain just synchronisation counters. As such, we do not go through a complete set of programming exercises. Instead, in this section we just discuss the important similarities and differences.

From the programming exercises used in Section 12.1, it is clear that one of the reasons a programmer might want to modify a guard in a derived class is to re-instantiate the same

[3] Other boolean operators, e.g., **or**, could be used to incrementally modify guards. However, it is rarely useful to do so.

generic synchronisation policy in a different way—to take into account either a new operation or an operation that has been re-implemented in such a way as to, say, change it from being a read-style operation to being a write-style operation.

For example, consider a guard that contains the expression:

exec(*list-of-operations*) = 0

We saw in Section 12.1 that a subclass could achieve the effect of adding operation *Foo* to *list-of-operations* in this guard by writing:

inherits and *exec*(Foo) = 0

Guards that contain scheduling predicates can be modified in a similar manner. For example, consider a guard that contains the predicate:

there_is_no(p **in** *waiting*(*list-of-operations*): *condition*)

A subclass could achieve the effect of adding operation *Foo* to *list-of-operations* in this guard by writing:

inherits and *there_is_no*(p **in** *waiting*(Foo): *condition*)

The code in Figures 12.10 and 12.11 shows a concrete example of this.

```
class Base {
  Bar(len: Int, ...) { ... }
synchronisation
  Bar: exec(Bar) = 0 and there_is_no(p in waiting(Bar): p.len < this_inv.len);
}
```

Figure 12.10: SJN[{*Bar*[len]}]

```
class Derived inherits Base {
  Foo(len: Int, ...) { ... }
synchronisation
  Foo, Bar: inherits Bar and exec(Foo) = 0
            and there_is_no(p in waiting(Foo): p.len < this_inv.len);
}
```

Figure 12.11: SJN[{*Foo*[len], *Bar*[len]}]

The base class implements the policy:

SJN[{*Bar*[len]}]

The derived class introduces a new operation, *Foo*, and modifies the synchronisation code so that it implements:

SJN[{*Foo*[len], *Bar*[len]}]

Just as we may want to modify a guard in order to re-instantiate the same generic policy in a different way, so too we might want to modify a guard in a derived class in order to instantiate a *different*, but related, generic synchronisation policy. An example of this was given in the 6th programming exercise of Section 12.1.2 when a derived class replaced the inherited *ReadersWriter* policy with *ReadersPriority*. While that particular example used just synchronisation counters, the same can be done with guards that contain scheduling predicates too.

For example, consider a base class that uses the following guard to implement a SJN policy upon operation *Foo*:

Foo: *exec*(Foo) = 0 **and** *there_is_no*(p **in** *waiting*(Foo): p.len < *this_inv*.len);

If a subclass wished to implement a SJN policy with FCFS sub-ordering then it could do so by re-implementing the guard anew, as follows:

Foo: *exec*(Foo) = 0 **and** *there_is_no*(p **in** *waiting*(Foo): p.len < *this_inv*.len
or p.len = *this_inv*.len **and** p.arr_time < *this_inv*.arr_time);

Alternatively, it could inherit the guard of the base class and incrementally modify it as follows:

Foo: **inherits and** *there_is_no*(p **in** *waiting*(Foo): p.len = *this_inv*.len
and p.arr_time < *this_inv*.arr_time);

12.2.1 Discussion

These examples show that the disadvantages that apply to the inheritance of guards containing synchronisation counters also apply to the inheritance of guards containing scheduling predicates, i.e.:

- Incremental modification of an inherited guard is likely to be as verbose (if not more so) than simply rewriting the guard anew. In fact, since the syntax of scheduling predicates is more verbose than that of synchronisation counters, this problem becomes more pronounced with scheduling predicates.
- It is also more difficult to write/understand guards that have been incrementally modified than guards that are written anew.
- Finally, this mechanism cannot always be employed. For example, while guards can often be incrementally modified to *add* a new operation to the instantiation of a generic synchronisation policy, the same can not be done in order to *remove* an operation.

12.3 Inheritance of Actions in Esp/Desp

A major enhancement in going from SP to Esp is that guards are complemented with the concept of actions. This then raises the question of how actions can be inherited and reused. In this section we discuss the support that Esp—actually Desp—provides for reuse of synchronisation code.

Note that the mechanism used to inherit synchronisation code in Desp is essentially the same as the inheritance mechanism used in the surveys so far. All that has changed is the syntax. For example, instead of the using the **inherits** keyword to obtain incremental change of a guard, Desp employs a "super" call to the same effect.

12.3.1 Inheritance of Actions

The following example shows that the inheritance of actions suffers from problems similar to those that beset the inheritance of guards.

The base class shown in Figure 12.12 and its support class in Figure 12.13 implement a starvation-free version of the SJN scheduler, i.e.:

SJN-Fair[{*Foo*[length]}]

Freedom from starvation is guaranteed by the *start*(Foo) action (method *s_Foo*). This action iterates over all the pending *Foo* requests and decrements the *length* of any that has been skipped over. Now consider a subclass that introduces a new operation, *Bar*, and modifies the synchronisation code to incorporate it, thus giving:

SJN-Fair[{*Foo*[length], *Bar*[length]}]

Code for this derived class is shown in Figure 12.14. The guard (method *g_Foo*) is incrementally modified in a manner similar to that of the previously discussed guard-based mechanisms, with the only difference being that of syntax: instead of using an **inherits** keyword, Desp invokes the guard of the parent class with the syntax for invoking a method in a parent class.

The *start* action (method *s_Foo*) in the derived class must iterate over not only pending *Foo* invocations but also pending *Bar* invocations. It achieves this by incremental modification: *s_Foo* invokes its namesake in the parent class in order to iterate over pending *Foo* invocations, and then uses a separate loop to iterate over pending *Bar* invocations.

The reader may be dismayed at the thought of having to use two loops instead of somehow being able to generalise the original loop so that it iterates over pending invocations for *Bar* as well as *Foo*. After all, this form of loop repetition increases the possibility of programming errors and the difficulty of code maintenance.

However, it should be noted that the way of incrementally modifying guards that contain synchronisation counters and/or scheduling predicates is quite similar to this loop repetition.

```
class Base inherits Any

public method Foo(length: Int)
begin ... end

synchronisation

method s_Foo(t: LengthInv)
var   p: LengthInv
begin
  for p in waiting(Foo) do
    if p.arr_time < t.arr_time then p.assign_length(p.length − 1); fi;
  od
end

method g_Foo(t: LengthInv): Bool
var   p: LengthInv
begin
  result := (exec(Foo) = 0)
            and (there_is_no(p in waiting(Foo): (p.length < t.length)
                                or ((p.length = t.length) and (p.arr_time < t.arr_time))));
end
map start(Foo) → s_Foo
map guard(Foo) → g_Foo
```

Figure 12.12: Starvation-free, Shortest Job Next scheduler (base class)

```
class LengthInv inherits Invocation

public var   length: Int

public method assign_length(val: Int)
begin
  length := val;
end
```

Figure 12.13: *LengthInv* class, used in the starvation-free SJN scheduler

As such, the incremental modification of guards that was discussed in Sections 12.1 and 12.2 should be looked upon with equal disfavour.

So as not to give a distorted appraisal of Esp, it should be noted that, in practice, not

```
class Derived inherits Base

public method Bar(length: Int)
begin ... end

synchronisation

method s_Foo(t: LengthInv)
var   p: LengthInv
begin
  super.s_Foo(t);
  for p in waiting(Bar) do
    if p.arr_time < t.arr_time then p.assign_length(p.length − 1); fi;
  od
end

method g_Foo(t: LengthInv): Bool
var   p: LengthInv
begin
  result := super.g_Foo(t) and (exec(Bar) = 0)
            and (there_is_no(p in waiting(Bar): (p.length < t.length)
                                  or ((p.length = t.length) and (p.arr_time < t.arr_time))));
end
map start(Bar) → s_Foo
map guard(Bar) → g_Foo
```

Figure 12.14: Starvation-free, Shortest Job Next scheduler (derived class)

all actions loop over pending invocations. For example, the Esp implementation of the Disk Head Scheduler (Figure 4.11) contains one guard, two actions and two other support routines. If this code were written as a Desp class and a derived class introduced a new operation, only a minimum of change would need to be made to the synchronisation code. To be specific, the guard would have to be re-implemented and the **map** construct used to ensure that the guard and existing actions are associated with the new operation. None of the code in the actions or the support routines would have to be changed in any way.

12.3.2 Wrapper Functions

Since the synchronisation mechanism of Desp has access to the full power of the data structures and flow-control constructs of the host language, one might consider that some of the other features of Desp might help with inheritance.

As an example, consider the following guards used to implement a bounded buffer:

Put: *exec*(Put, Get) = 0 **and** *term*(Put) − *term*(get) < Size;
Get: *exec*(Put, Get) = 0 **and** *term*(Put) − *term*(get) > 0;

One could write two "wrapper functions"—*mutex* and *num*—as follows:

method mutex: Bool **begin** result := *exec*(Put, Get) = 0; **end**
method num: Int **begin** result := *term*(Put) − *term*(Get); **end**

Then the guards for the buffer could be rewritten as:

Put: mutex **and** num < Size;
Get: mutex **and** num > 0;

This not only helps make the guards easier to understand but also (apparently) helps somewhat with the inheritance of guards in derived classes. For example, consider a subclass that introduces a new operation, *PutFront*. Although the *implementation* of the wrapper functions will need to be changed to take into account this new operation, the *concepts* they embody still remain valid and so the guards which use them will not have to be changed.

Functions might be used as wrappers not only around synchronisation counters but also around collections of invocations. For example, Figure 12.15 shows an implementation of the starvation-free version of the SJN scheduler. The difference between this version and that shown previously in Figure 12.12 is that this version uses two wrapper functions—one for *exec*(Foo) and another for *waiting*(Foo). These wrapper functions are then used in the action and guard. Figure 12.16 shows a derived class which introduces a new operation, Bar. The only changes required to the synchronisation code required are (i) to **map** the existing guard and action onto the new operation, and (ii) to re-implement the wrapper functions.

However, the use of wrapper functions does not usually result in a decrease of the number of lines that a programmer must write. This is because the amount of code required for the wrapper functions often equals or even exceeds the amount of code in guards and actions.

For example, consider the following guards which implement the readers priority variant of the readers/writer policy:

Read: *exec*(Write) = 0;
Write: *exec*(Read, Write) = 0 **and** *wait*(Read) = 0;

To implement this with the aid of wrapper functions in an attempt to aid reuse would require three functions: one for each of *exec*(Read), *exec*(Write) and *wait*(Read).

The problem with using wrapper functions is that the abstraction they provide is at too low a level. Rather than abstracting at the level of, say, *exec*(*list-of-operations*) and *waiting*(*list-of-operations*), programmers should be able to abstract at the higher level of *list-of-operations* itself and then be able to apply *wait*, *exec* and *waiting* and so on to this abstraction. This is, of course, the level of abstraction that generic synchronisation policies provide. Later in this chapter we will examine how the use of generic synchronisation helps to reduce the harmful effects of the ISVIS conflict.

```
class Base inherits Any

public method Foo(length: Int)      begin ... end

synchronisation

method waiting_Foo: Collection[LengthInv]   begin   result := waiting(Foo);   end
method exec_Foo: Int   begin   result := exec(Foo);   end

method s_Foo(t: LengthInv)
var   p: LengthInv
begin
  for p in waiting_Foo do
    if p.arr_time < t.arr_time then p.assign_length(p.length − 1); fi;
  od
end

method g_Foo(t: LengthInv): Bool
var   p: LengthInv
begin
  result := (exec_Foo = 0) and (there_is_no(p in waiting_Foo: (p.length < t.length)
                                  or ((p.length = t.length) and (p.arr_time < t.arr_time))));
end
map start(Foo) → s_Foo
map guard(Foo) → g_Foo
```

Figure 12.15: Starvation-free, SJN scheduler with wrapper functions (base class)

```
class Derived inherits Base

public method Bar(length: Int)      begin ... end

synchronisation

method waiting_Foo: Collection[LengthInv]   begin   result := waiting(Foo Bar);   end
method exec_Foo: Int   begin   result := exec(Foo Bar);   end

map start(Bar) → s_Foo
map guard(Bar) → g_Foo
```

Figure 12.16: Starvation-free, SJN scheduler with wrapper functions (derived class)

12.4 Inheritance of Synchronisation Code in Guide

In this section, we do not consider any of the proposed extensions presented at workshops [RR92, Riv92], but instead confine the discussion to a more stable version of Guide [DDR+91]. This version of Guide allows the following constructs to appear in guards:

- Synchronisation counters
- Parameters. However, recall that parameters of different invocations cannot be compared with each other; rather, parameters can be compared only with, say, instance variables or constants.
- Instance variables.

Section 12.1 has already discussed the inheritance issues concerning synchronisation counters; and the limited access to parameters is not interesting so this section just discusses the inheritance issues that arise when guards have access to instance variables.

As explained in Chapter 3, instance variables that are accessed in guards are really synchronisation variables that just happen to be implemented as instance variables (due to lack of language support for synchronisation variables). Thus, we can expect that if a derived class changes its inherited synchronisation policy in a manner that requires the maintenance of a synchronisation variable then this will be achieved by implementing it as an instance variable and modifying the inherited sequential code in order to maintain this variable.

An (admittedly contrived) example to illustrate this is provided by the following programming exercise: inherit from a bounded buffer class and introduce a new operation, *Gget*, that cannot execute immediately after an invocation of *Put*. Implementations of the base and derived classes are shown in Figures 12.17 and 12.18, respectively. Note that the inherited operations have to be incrementally modified in order to maintain the variable *after_put*.

```
class Buffer[elem, Size] {
  int   num;
  ... //other instance variables
  Buffer(...) { ... } // initialisation
  Put(...) { ... }
  Get(...) { ... }
synchronisation
  Put: exec(Put, Get) = 0 and num < Size;
  Get: exec(Put, Get) = 0 and num > 0;
}
```

Figure 12.17: A bounded buffer

```
class ExtBuffer[elem, Size] inherits Buffer {
  Bool    after_put;
  ExtBuffer(...) { super.Buffer(...); after_put := false; }
  Put(...) { super.Put(...); after_put := true; }
  Get(...) { super.Put(...); after_put := false; }
  Gget(...) { ...; after_put := false; }
synchronisation
  Put, Get: inherits and exec(Gget) = 0;
  Gget: inherits Get and exec(Gget) = 0 and not after_put;
}
```

Figure 12.18: A bounded buffer with operation *Gget*

12.5 Summary of Inheritance in Guard-based Mechanisms

So far we have examined how well various guard-based synchronisation mechanisms promote reuse of inherited synchronisation code through incremental modification. The results are not encouraging. It is often (i) shorter, and (ii) easier to rewrite guards anew than it is to incrementally modify them. Furthermore, (iii) this method of reuse cannot always be used, e.g., it can be used to add new constraints onto guards but cannot be used to remove or replace existing constraints.

If there is any good news it is that, with one exception, these guard-based mechanisms keep synchronisation code separate from sequential code, and thus poor support for reuse of inherited synchronisation code does not hinder the reuse of inherited sequential code. The exception is Guide which implements synchronisation variables as (sequential) instance variables. Even here, the interference is not too severe: if the synchronisation code of a derived class wishes to declare/maintain a new synchronisation variable then the required changes to the sequential code can likely be made via incremental modification to the operations rather than re-implementing them anew.

In the surveys carried out so far, the synchronisation mechanisms have had different amounts of expressive power but all have used the same inheritance mechanism. This was why the results of the surveys have been so similar. The next section examines a different inheritance mechanism, and shows that its effectiveness in tackling the inheritance anomaly is quite different.

12.6 Inheritance of Enabled-sets in Rosette

Since the Rosette language does not support internal concurrency,[4] we use the ISVIS matrix shown in Figure 11.3 to evaluate it. This lack of internal concurrency means that we cannot use programming exercises based on readers/writer in order to evaluate the inheritance mechanism of Enabled-sets. Instead, many of the examples will be based on a bounded buffer. Recall from Section 11.5.2 that this varying of programming exercises for different synchronisation mechanisms is necessary to ensure that the evaluation of an inheritance mechanism is not negatively biased due to limited expressive power of a host synchronisation mechanism.

12.6.1 1st Column of the Matrix

We start off with some programming exercises that test the first column of the ISVIS matrix.

```
class Base {
  ... // instance variables
  state empty enables {Put};
  state full enables {Get};
  state partial enables empty.ops + full.ops;
  Base(...)
  { ...   // sequential initialisation
    become empty;   // buffer is empty initially
  }
  Put(...)
  { "sequential code to place item at end of buffer";
    if "buffer is full" then become full; else become partial; endif;
  }
  Get(...)
  { "sequential code to remove item from front of buffer";
    if "buffer is empty" then become empty; else become partial; endif;
  }
}
```

Figure 12.19: 1st programming exercise (Rosette)—base class

[4] The statement that Rosette does not support internal concurrency is slightly misleading. In Rosette when an operation is permitted to start execution, it has exclusive access to the object, i.e., it executes in mutual exclusion. However, when it executes a **become** statement then a copy of the object is made. The currently executing operation will work on its own copy of the object and the other copy will be used by the next invocation that executes. Thus any updates to instance variables after executing a **become** statement will not be reflected in the object that is accessed by future invocations. For this reason, the **become** statement usually appears at, or near, the very end of an operation. Thus, in effect, there is little chance for internal concurrency.

1st Programming Exercise—cell(2a, 1)

Write a synchronised bounded buffer class that contains operations *Put* and *Get*. Inherit from this class and do the following:

Totally re-implement operation *Put*.

The code of the base and derived classes are shown in Figure 12.19 and 12.20.

```
class Derived inherits Base {
  Put(...)
  { "new sequential code to place item at end of buffer";
    if "buffer is full" then become full; else become partial; endif;
  }
}
```

Figure 12.20: 1st programming exercise (Rosette)—derived class

Note that since the synchronisation code to determine what set of operations should be "enabled" next is placed inside the bodies of operations, changing the sequential code of an operations necessitates a retyping of any such synchronisation code inside that operation, even though it may not have changed. (This can be seen in the re-implementation of operation *Put* in the derived class.) This is a very clear example of the modification of inherited sequential code hindering the reuse of inherited synchronisation code.

2nd Programming Exercise—cell(2b, 1)

Utilising the base class from the previous exercise, inherit and do the following in the derived class:

Incrementally modify operation *Put*.

Let us consider two possible cases. The first is when the operation in the subclass is written in the form:

```
Put(...) {
  "new sequential code"
  super.Put(...);
}
```

The second case is when the new sequential code is placed *after* the "super" call. The first case works fine, while the second one does not since the new sequential code will be executed *after* the synchronisation code in the "super" call and, as said at the start of Section 12.6, synchronisation code should appear at the end of an operation. Thus, in the second case the operation will have to be re-implemented anew. Hence we see that the inheritance of synchronisation code, without the code even being modified, can hinder the reuse of inherited sequential code.

3rd Programming Exercise—cell(3c, 1)

As before, write a synchronised bounded buffer with operations *Put* and *Get*, and the following SPC:

BBuf[{Put}, {Get}, Size]

Inherit from this class and do the following:

Introduce a new put-style operation, *PutFront*.

Obviously this will require that the synchronisation code be changed so that it implements:

BBuf[{Put, PutFront}, {Get}, Size]

The base class has already been shown (Figure 12.19); the derived class is shown in Figure 12.21. The *empty* enabled-set is incrementally modified to include the new operation. In doing this, the rest of the inherited synchronised code takes account of this new operation.

```
class Derived inherits Base {
  state empty enables super.empty.ops + {PutFront};
  PutFront(...)
  { "sequential code to place item at front of buffer";
    if "buffer is full" then become full; else become partial; endif;
  }
}
```

Figure 12.21: 3rd programming exercise (Rosette)—derived class

While the incremental modification of the *empty* enabled-set is laudable, one complaint we have concerns the new operation *PutFront*. The synchronisation code at the end of this operation is the same as that at the end of the *Put* operation. Such repetition of code is error prone; it would be better if this identical synchronisation code could be written once and then shared by all put-style operations. We discuss this possibility further in Section 12.6.3.

4th Programming Exercise—cell(3c, 1)

As before, write a synchronised bounded buffer with operations *Put* and *Get*, and the following SPC:

BBuf[{Put}, {Get}, Size]

Inherit from this class and do the following:

Introduce a new operation, *Get2*, that removes two items from the buffer.

Obviously this will require that the synchronisation code be changed so that it implements, say:

Another-BBuf[{Put}, {Get}, {Get2}, Size]

The difference between this programming exercise and the previous one is that, here, the introduction of a new operation causes the derived class to instantiate a *different* generic synchronisation policy, while in the previous exercise the derived class re-instantiated the *same* generic synchronisation policy.

The base class (shown previously in Figure 12.19) defined three enabled-sets: *empty*, *full* and *partial*. *Get2* does not fit neatly into these existing sets. In particular, the concept of a partially full buffer will need to be subdivided into: (i) a buffer containing one item, and (ii) a buffer containing two or more items. The *Get2* operation cannot execute in the former state but can in the latter.

Obviously the synchronisation code at the end of the existing operations—*Put* and *Get*—will have to be changed in order to take into account these new states. In order to make such changes we have to rewrite these operations anew. Thus we see that introducing a new operation that requires a *different* generic synchronisation policy to be instantiated might make it impossible to reuse *any* of the inherited sequential operations.

12.6.2 2nd Column of the Matrix

We now use some programming exercises to test the second column of the ISVIS matrix.

5th Programming Exercise—cell(1, 2)

Write a class that contains two operations, A and B, and the following SPC:

Priority[{A}, {B}]

Inherit from this class and change the synchronisation code so that the priority of the two operations are swapped, i.e., the new SPC is:

Priority[{B}, {A}]

The code for this exercise is shown in Figure 12.22. Note that this code uses a small extension of the Enabled-sets mechanism as previously discussed. The keyword **enables-by** [TS89, pg. 110] takes a list of sets of operations and enables operations in the order that these sets are specified. Thus in this example operation A has priority over B. The only change the subclass needs to make is to redefine the *priority* enable set in order to swap the priorities of the operations.

6th Programming Exercise—cell(1, 2)

As in the previous exercise, write a class that contains two operations, A and B, and the following SPC:

Priority[{A}, {B}]

```
class Base {
  state priority enables-by {A}, {B};
  Base(...)
  { ...   // sequential initialisation
    become priority;
  }
  A(...)
  { "sequential code";
    become priority;
  }
  B(...)
  { "sequential code";
    become priority;
  }
}
```

```
class Derived inherits Base {
  state priority enables-by {B}, {A};
}
```

Figure 12.22: 5th programming exercise (Rosette)

Inherit from this class and change the synchronisation code so that these two operations execute in strict alternation, i.e., the new SPC is:

Alternation[{A}, {B}]

This exercise is similar to the previous one in that the synchronisation code changes in a manner that should not affect the sequential code. However, there is an important difference: the previous exercise re-instantiated the *same* generic synchronisation policy while in this exercise the subclass instantiates a *different* policy. The net effect is similar to that in the fourth programming exercise: the inherited operations have to be re-implemented anew in order to change the synchronisation code contained in them. This can be easily seen in in Figure 12.23.

7th Programming Exercise—cell(2, 2)

The first and second programming exercises show that changes to sequential code hinders reuse of synchronisation code. Similarly, the sixth programming exercise show that changes to synchronisation code can hinder reuse of sequential code. When both types of changes are combined together, as in this cell, there is unlikely to be any possibility for reuse of inherited code. Interested readers can devise a programming exercise to verify this for themselves.

8th Programming Exercise—cell(3c, 2)

Write a class that contains two operations, A and B, and the following SPC:

Priority[{A}, {B}]

```
class Base {
  state priority enables-by {A}, {B};
  Base(...)
  { ...   // sequential initialisation
    become priority;
  }
  A(...)
  { "sequential code";
    become priority;
  }
  B(...)
  { "sequential code";
    become priority;
  }
}
```

```
class Derived inherits Base {
  state first enables {A};
  state second enables {B};
  Derived(...)
  { ...   // sequential initialisation
    become first;
  }
  A(...)
  { "sequential code";
    become second;
  }
  B(...)
  { "sequential code";
    become first;
  }
}
```

Figure 12.23: 6th programming exercise (Rosette)

Inherit from this class and introduce a new operation, C, that has a priority similar to that of A. Also change the synchronisation code so that the priority of the two sets of operations are swapped, i.e., the new SPC is:

Priority[{B}, {A, C}]

The code for this exercise is shown in 12.24. This cell is quite similar to cell(3c, 1) in that the synchronisation code can be easily changed to accommodate a new operation if doing so involves re-instantiating the *same* generic policy. However, if the change in synchronisation code had involved instantiating a *different* policy, e.g., *Alternation*, then it would not have been possible to reuse inherited code.

12.6.3 Discussion

As discussed in Section 10.1.5, Enabled-sets are a form of generic synchronisation policies. Of particular relevance to the present discussion is that a subclass adding new operations to an inherited enabled-set is the means by which a subclass re-instantiates the same generic synchronisation policy as used in its parent class. Examples of this can be seen in the 3rd, 5th and 8th programming exercises on pages 190, 191 and 192, respectively. Enabled sets handles this re-instantiation of a policy gracefully. However, if a subclass wishes to instantiate a generic synchronisation policy that is *different* to that in its parent class then this requires that that all the operations be redefined. Examples of this can be seen in the 4th and 6th programming exercises on pages 190 and 191, respectively.

```
class Base {
  state priority enables-by {A}, {B};
  Base(...)
  { ...   // sequential initialisation
    become priority;
  }
  A(...)
  { "sequential code";
    become priority;
  }
  B(...)
  { "sequential code";
    become priority;
  }
}
```

```
class Derived inherits Base {
  state priority enables-by {B}, {A, C};
  C(...)
  { "sequential code";
    become priority;
  }
}
```

Figure 12.24: 8th programming exercise (Rosette)

The cause of this hindrance to reuse is the lack of separation between synchronisation code and sequential code. This lack of separation also hinders reuse when a subclass wishes to change the sequential code of an operation, as in the 1st and 2nd programming exercises on pages 189 and 189, respectively.

One way to considerably lessen the hindrances to reuse would be to syntactically separate sequential code and synchronisation code from one another. For example, if the restriction is made that synchronisation code can only appear at the end of an operation (which, in practice, is the case anyway) then a *term* action could be introduced and the synchronisation code placed there, thus leading to a syntactic separation of code such as:

```
Put(...) { "sequential code"; }
term(Put) → { "synchronisation code"; }
```

Such a separation would enable the sequential code to be inherited and modified without the need to retype the synchronisation code, and vice versa.

However, this can be improved further. The synchronisation code at the end of all put-style operations will be identical; also, the synchronisation code at the end of get-style operations is quite similar, though not identical, to that at the end of put-style operations. It seems feasible that the synchronisation code at the end of *all* these operations could be generalised and factored out into a single *term* action. Figure 12.25 illustrates this by example. The run-time system will automatically invoke the *term* action (represented here as an operation) at the end of any operation such as *Put* or *Get*, and it will determine which set of operations will be "enabled" next.

The code of this class is approximately the same length as that of the original base

```
class Base {
  ... // instance variables
  Put(...) { "sequential code to place item at end of buffer"; }
  Get(...) { "sequential code to place item at end of buffer"; }
private:
  state empty enables {Put};
  state full enables {Get};
  state partial enables empty.ops + full.ops;
  term()
  { if "buffer is full" then become full;
    else if "buffer is empty" then become empty;
    else become partial;
    endif;
  }
}
```

Figure 12.25: Bounded buffer with modified version of Rosette—base class

class (Figure 12.19). However, the benefits of separating the synchronisation code from the sequential code can be seen in derived classes. If a derived class wishes to, say, totally re-implement operation *Put* then it can do so without having to retype any synchronisation code (1st programming exercise). Similarly for a derived class that wishes to incrementally modify *Put* (2nd programming exercise), or introduce a new put-style operation (3rd programming exercise). If a derived class wishes to instantiate a generic synchronisation policy different to that of its parent class (e.g., as in the 4th programming exercise) then it can change the synchronisation code without affecting the inherited sequential code.

Note that this separation of synchronisation code from sequential code does not affect the synchronisation mechanism *per se*. Rather it can be considered to be a change in the inheritance mechanism since the only affect it has is to facilitate inheritance of code.

12.7 Inheritance of Enabled-sets in Arche

Like Rosette, Arche [BI92] uses enabled-sets and the inheritance of synchronisation code in both is similar in many respects. As such, we will not work through a complete set of programming exercises to evaluate Arche, but rather just discuss the similarities and differences between the two languages.

In Arche, several **become** statements can be executed in an operation. If this happens then only the last one executed before the end of the operation will take effect. This relaxation on the usage of the **become** statement reduces the harmful effects of the ISVIS conflict in Arche, as we now discuss.

If a subclass instantiates a generic synchronisation policy different to that of its parent class then this means that along with defining new enabled-sets, the synchronisation code inside the bodies of operations needs to be changed too. The 4th and 6th programming exercises used to evaluate Rosette (on pages 190 and 191, respectively) are examples of this, and in these the operations had to be re-implemented anew. Arche, however, can handle this situation more gracefully by incrementally modifying the operations to add on the new synchronisation code. This works because if several **become** statements are executed in an operation then only the last one takes effect.

For example, a subclass might change the synchronisation code in an inherited operation as follows:

```
Foo(...) {
  super.Foo(...);
  "new synchronisation code"
}
```

Likewise, if a subclass wishes to incrementally modify the sequential code of an inherited operation then it can do so as follows:

```
Foo(...) {
  super.Foo(...);
  "new sequential code"
}
```

We were unable to do this in Rosette due to its restriction that the **become** statement should appear at the end of an operation. However, since Arche relaxes this restriction, a **become** statement can be executed via the call to *Foo* in the parent class and then more sequential code executed afterwards.

In effect, Arche uses the same synchronisation mechanism as Rosette, but a different inheritance mechanism—one which reduces the harmful effects of the ISVIS conflict. However, Arche is not without problems. In particular, if a subclass wishes to totally re-implement the sequential code of an operation then it must also re-implement the synchronisation code of that operation since it is impossible to reuse the synchronisation code of an operation without also reusing its sequential code.

12.8 Inheritance of Synchronisation Code in Eiffel||

The Eiffel|| language does not support concurrency within objects. As such, we use the ISVIS matrix shown in Figure 11.3 for evaluation.

12.8.1 1st Column of the Matrix

We start off with some programming exercises that test the first column of the ISVIS matrix.

1st Programming Exercise—cell(2, 1)

Write a synchronised class that contains operations A and B, that execute in strict alternation, i.e., the SPC of this class is:

Alternation[{A}, {B}]

Inherit from this class and make the following changes:

- Totally re-implement operation A. (This tests cell(2a, 1).)
- Incrementally modify operation *Put*. (This tests cell(2b, 1).)

The code for this exercise is shown in Figure 12.26. In this case we can see that the modification of the sequential code does not hinder the reuse of the inherited synchronisation code. Note that the *serve_oldest* operation is non-blocking, i.e., if it cannot find a suitable invocation to serve then it returns immediately. As such, it is preceded by a call to *wait_for_invocation_of*.

```
class Base inherits PROCESS {
  A(...) { ... }
  B(...) { ... }
  Live()
  { while true do
      wait_for_invocation_of(A);
      serve_oldest(A);
      wait_for_invocation_of(B);
      serve_oldest(B);
    end
  }
}
```

```
class Derived inherits Base {
  A(...) { ... }
  B(...) { super.B(...); ... }
  // no change to Live
}
```

Figure 12.26: 1st programming exercise (Eiffel||)

2nd Programming Exercise—cell(3c, 1)

As in the previous example, write a synchronised class that contains operations A and B, that execute in strict alternation, i.e., the SPC of this class is:

Alternation[{A}, {B}]

Inherit from this class and introduce a new operation, C. This necessitates a change in the SPC so change it to:

Alternation[{A}, {B, C}]

The base class is as in the previous exercise (shown in Figure 12.26). The derived class is shown in Figure 12.27. As can be seen, the synchronisation code had to be re-implemented anew in order to accommodate the new operation. In general, Eiffel|| does not provide any means of incrementally modifying synchronisation code.

One exception to this is the default synchronisation policy provided by the implementation of *Live* in the class *PROCESS*. This policy is written in a manner that is independent of the number of operations upon which it is instantiated. Thus, this implementation of *Live* could implement "FCFS[{A, B}]" in a base class and "FCFS[{A, B, C}]" in a subclass that introduces a new operation, C.

```
class Derived inherits Base {
  C(...) { ... }
  Live()
  { while true do
      wait_for_invocation_of(A);
      serve_oldest(A);
      wait_for_invocation_of(B, C);
      serve_oldest(B, C);
    end
  }
}
```

Figure 12.27: 2nd programming exercise (Eiffel||)—derived class

12.8.2 2nd Column of the Matrix

We now move onto the second column of the ISVIS matrix.

3rd Programming Exercise—cell(1, 2)

Write a synchronised class that contains operations, A and B, and gives priority to servicing invocations of A, i.e., the SPC for the class is:

Priority[{A}, {B}]

Inherit from this class and swap the priority of the operations, i.e., change the SPC to be:

Priority[{B}, {A}]

The base class for this exercise is shown in Figure 12.28. The only change that the derived class needs to make is to replace references to operation A with B and vice versa. However, as in the previous exercise, the only way this can be done is to re-implement the synchronisation code anew.

```
class Base inherits PROCESS {
  A(...) { ... }
  B(...) { ... }
  Live()
  { while true do
      wait_for_invocation_of(A, B);
      if exist_invocation_of(A) then serve_oldest(A);
      else serve_oldest(B);
      endif
    end
  }
}
```

Figure 12.28: 3rd programming exercise (Eiffel||)—base class

An alternative programming exercise for this cell would be for the subclass to instantiate a different generic policy rather than re-instantiate the same policy as used in the parent class. For example, the SPC might change from a *Priority* policy to an *Alternation* policy. The results would be the same, i.e., changing the policy would not hinder the reuse of inherited sequential code, and the synchronisation code would have to be re-implemented anew rather than there being a possibility to incrementally modify the synchronisation code inherited from the parent class.

4th Programming Exercise—cell(2, 2)

As in the previous exercise, write a synchronised class that contains operations, A and B, and gives priority to servicing invocations of A, i.e., the SPC for the class is:

Priority[{A}, {B}]

Inherit from this class and swap the priority of the operations, i.e., change the SPC to be:

Priority[{B}, {A}]

Also, totally re-implement operation A, thus testing cell(2a, 2), and incrementally modify operation B, thus testing cell(2b, 2).

The code for this exercise is quite similar to that of the previous one, with the addition that the sequential operations have been changed. The changes to the sequential code do not affect the synchronisation code, nor vice versa.

5th Programming Exercise—cell(3c, 2)

Write a synchronised class that contains operations, A and B, and gives priority to servicing invocations of A, i.e., the SPC for the class is:

Priority[{A}, {B}]

Inherit from this class and make the following changes:

- Change the synchronisation code so that it implements an *Alternation* policy.
- Introduce a new operation, C, the synchronisation constrains of which will be similar to that of B.

The base class is the same as in the third programming exercise (Figure 12.28). The derived class is shown in Figure 12.29. As can be seen, the introduction of the new operation forces a rewrite of the synchronisation code but this is obtained for free since the synchronisation policy is being changed anyway.

```
class Derived inherits Base {
  C(...) { ... }
  Live()
  { while true do
      wait_for_invocation_of(A);
      serve_oldest(A);
      wait_for_invocation_of(B, C);
      serve_oldest(B, C);
    end
  }
}
```

Figure 12.29: 5th programming exercise (Eiffel||)—base class

12.8.3 Discussion

In Eiffel||, the synchronisation code of a class is placed in the *Live* operation rather than being spread out among all the operations of the class as in the Enabled Sets-based mechanisms. The advantage of this separation is clear: a re-implementation of *Live* does not necessitate a re-implementation of the sequential operations, and vice versa. (Of course, the introduction of a new operation will necessitate a change of the synchronisation code.) The main draw-back of Eiffel|| is that it does not provide any means to make incremental modifications to synchronisation code. If *any* change is made to *Live* then it must be re-implemented anew. This is in contrast to the guard-based mechanisms and the Enabled Sets-based mechanisms, both of which make an attempt (albeit not an entirely successful one) to permit synchronisation code to be incrementally modified.

12.9 Generic Synchronisation Policies

This chapter has employed the ISVIS matrix to evaluate how various inheritance mechanisms for synchronisation code conflict with the inheritance of sequential code. Throughout these evaluations the concept of generic synchronisation policies has been used as a means of *discussing* the ISVIS conflict. However, as we have shown in Part III of this thesis, it is possible to provide language support for GSPs. As such, we now evaluate how the use of GSPs might help in tackling the ISVIS conflict.

12.9.1 1st Column of the Matrix

We start off with some programming examples to test the cells in the first column of the ISVIS matrix.

1st Programming Exercise—cell(2a, 1)) and cell(2b, 1)

Write a synchronised class that contains operations A, B and C, and the following synchronisation policy:

> ReadersWriter[{A, B}, {C}]

Inherit from this class and do the following:

- Totally re-implement A, keeping it as a read-style operation. (This tests cell(2a, 1)).
- Incrementally modify B, keeping it as a read-style operation. (This tests cell(2b, 1)).

The code of the base and derived classes is shown in Figure 12.30. In this case we can see that, as expected, the modification of the sequential code did not hinder the reuse of the inherited synchronisation code.

```
class Base {
  A(...) { ... }  B(...) { ... }
  C(...) { ... }
synchronisation
  ReadersWriter[ {A, B}, {C} ]
}
```

```
class Derived inherits Base {
  A(...) { ... }
  B(...) { super.B(...); ... }
}
```

Figure 12.30: 1st Programming exercise (generic synchronisation policy)

2nd Programming Exercise—cell(3c, 1)

Using the same base class as in the previous programming exercise, inherit and make the following change to the derived class:

> Introduce a new write-style operation, D.

Introducing this new operation violates the inherited SPC and so the following new one is adopted:

ReadersWriter[{A, B}, {C, D}]

The code for the derived class is shown in Figure 12.31 (the base class is as previously shown in Figure 12.30). The change to the sequential code has necessitated a change to the synchronisation code, which is achieved in just a single line of code: by re-instantiating the same generic policy. Note that this re-instantiation can be written anew or written as an incremental modification of the instantiation of the policy in the parent class. (Currently, only the former is supported in GASP.)

```
class Derived inherits Base {
  D(...) { ... }
synchronisation
  ReadersWriter[ {A, B}, {C, D} ]
  // alternatively...
  // super.policy[ super.ReadOps, super.WriteOps + {D} ]
}
```

Figure 12.31: 2nd Programming exercise (generic synchronisation policy)—derived class

3rd Programming Exercise—cell(3a, 1) and cell(3b, 1)

Using the same base class as in the previous programming exercises, inherit and make the following change to a derived class:

Totally re-implement *B* in a manner that makes it become a write-style operation.

Making this change violates the inherited SPC and so the following new one is adopted:

ReadersWriter[{A}, {B, C}]

As in the previous exercise, the re-instantiation of the generic synchronisation policy takes just a single line. The derived code is shown in Figure 12.32 (the base class is as previously shown in Figure 12.30).

This programming exercise involved a total re-implementation of *B* and hence tested cell(3a, 1); however, it is obvious that if *B* had been incrementally modified (thus testing cell(3b, 1)) then the same results would have been obtained.

12.9.2 2nd Column of the Matrix

We now move on to test the second column of the ISVIS matrix in which changes are made to the synchronisation code of a subclass that, by themselves, do not necessitate any change to the inherited sequential code.

```
class Derived inherits Base {
  B(...) { ... }
synchronisation
  ReadersWriter[ {A},  {B, C} ]
  // alternatively...
  // super.policy[ super.ReadOps − {B},  super.WriteOps + {B} ]
}
```

Figure 12.32: 3rd Programming exercise (generic synchronisation policy)

4th Programming Exercise—cell(1, 2)

Write a synchronised base class that contains operations A, B and C, and the following SPC:

ReadersWriter[{A, B}, {C}]

Inherit from this class and in the derived class change the SPC to be:

ReadersPriority[{A, B}, {C}]

The code for the base and derived classes are shown in Figures 12.33 and Figure 12.34, respectively.

```
class Base {
  A(...) { ... }  B(...) { ... }
  C(...) { ... }
synchronisation
  ReadersWriter[ {A, B},  {C} ]
}
```

Figure 12.33: 4th Programming exercise (generic synchronisation policy)—base class

```
class Derived inherits Base {
  // no change to seq code
synchronisation
  ReadersPriority[ {A, B},  {C} ]
  // alternatively...
  // ReadersPriority[ super.ReadOps,  super.WriteOps ]
}
```

Figure 12.34: 4th Programming exercise (generic synchronisation policy)—derived class

As can be seen, the change in the synchronisation code does not hinder the inheritance of the sequential code. Also, even though the subclass uses a different generic policy to that

in the parent class, the instantiation of the policy in the subclass can be expressed terms of the instantiation of the policy in the parent class.

5th Programming Exercise—cell(2, 2)

This exercise is similar to the previous one, except that the derived class makes some changes to the sequential code as well as to the synchronisation code. The changes to the sequential code do not violate the inherited SPC and hence should not necessitate changes to the synchronisation code.

Write a synchronised base class that contains operations A, B and C, and the following SPC:

ReadersWriter[{A, B}, {C}]

Inherit from this class and in the derived class make the following changes:

- Change the synchronisation code so that it implements:

  ReadersPriority[{A, B} {C}]

- Totally re-implement A, keeping it as a read-style operation. (This tests cell(2a, 2)).
- Incrementally modify B, keeping it as a read-style operation. (This tests cell(2b, 2)).

There is little point in showing the code for the derived class since it is similar to that of the previous exercise (the code of which is shown in Figure 12.34), except that operations A and B are modified as mentioned above. This modification of the sequential operations does not hinder the inheritance of synchronisation code at all.

6th Programming Exercise—cell(3, 2)

As in the two previous examples, the derived class instantiates a generic policy different to that used in the base class. This change in policy should not affect the sequential code. However, the programmer makes some changes to the sequential code anyway—changes that would have necessitated change in the synchronisation code even if a programmer had not planned on changing the synchronisation code.

Write a synchronised base class that contains operations A, B and C, and the following SPC:

ReadersWriter[{A, B}, {C}]

Inherit from this class and in the derived class make the following changes:

- Change the synchronisation code so that it implements a *ReadersPriority* policy.
- Totally re-implement C, and in doing so change it into a read-style operation. (This tests cell(3a, 2)).

- Incrementally modify A, and in doing so change it into a write-style operation. (This tests cell(3b, 2)).
- Introduce a new write-style operation, D. (This tests cell(3c, 2)).

The code for the derived class is shown in Figure 12.35 (the base class is as previously shown in Figure 12.33).

```
class Derived inherits Base {
  A(...) { super.A(...); ...}
  C(...) { ... }
  D(...) { ... }
synchronisation
  ReadersPriority[ {B, C},  {A, D} ]
  // alternatively...
  // ReadersPriority[ super.ReadOps − {A} + {C},  super.WriteOps − {C} + {A, D} ]
```

Figure 12.35: 6th Programming exercise (generic synchronisation policy)—derived class

As before, the change from the basic *ReadersWriter* policy to *ReadersPriority* does not have any effect on the sequential code. However, the changes made to the sequential code *do* affect the synchronisation code, to such an extent that it is shorter to express the instantiation of the policy in the subclass anew rather than as an incremental modification of the policy in the parent class.

12.9.3 3rd Column of the Matrix

We now move on to test the third column of the ISVIS matrix in which changes are made to the synchronisation code of a subclass that necessitate change to the inherited sequential code.

7th Programming Exercise—cell(1, 3)

Write a base class that contains operations A, B and C, all of which update instance variables and hence execute in mutual exclusion. The SPC of this class is:

Mutex[{A, B, C}]

Inherit from this class and change the synchronisation policy to *ReadersWriter*, with A treated as a read-style operation, in order to permit more internal concurrency. This changes the SPC to:

ReadersWriter[{A}, {B, C}]

This, of course, necessitates that operation A be re-implemented to ensure that it does not update any instance variables. The code for the base and derived classes are shown in Figures 12.36 and 12.37, respectively.

```
class Base {
   A(...) { ... }
   B(...) { ... }
   C(...) { ... }
synchronisation
   Mutex[ {A, B, C} ]
}
```

Figure 12.36: 7th Programming exercise (generic synchronisation policy)-base class

```
class Derived inherits Base {
   A(...) { ...}
synchronisation
   ReadersPriority[ {A},   {B, C} ]
   // alternatively...
   // ReadersPriority[ {A},   super.Ops − {A} ]
```

Figure 12.37: 7th Programming exercise (generic synchronisation policy)—derived class

8th Programming Exercise—cell(2, 3)

As in the previous exercise, write a base class that contains operations A, B and C, all of which update instance variables and hence execute in mutual exclusion, i.e., the SPC of the class is:

Mutex[{A, B, C}]

Inherit from this class and in the derived class change the synchronisation policy to *ReadersWriter*, with A treated as a read-style operation, in order to permit more internal concurrency. This changes the SPC to:

ReadersWriter[{A},   {B, C}]

As before this necessitates that operation A be re-implemented to ensure that it does not update any instance variables. Make the following additional changes to the sequential code:

- Totally re-implement B in a manner that keeps it as a write-style operation. (This tests cell(2a, 3).)

- Incrementally modify C in a manner that keeps it as a write-style operation. (This tests cell(2b, 3).)

The code for this exercise is similar to that of the previous exercise (shown in Figures 12.36 and 12.37). The only difference is that the derived class re-implements operations B and C. These additional changes to the sequential code do not affect the synchronisation code.

9th Programming Exercise—cell(3, 3)

As before, write a base class that contains operations A, B and C, all of which update instance variables and thus execute in mutual exclusion, i.e., the SPC of the class is:

Mutex[{A, B, C}]

Inherit from this class and make the following changes:

- Change the synchronisation policy to *ReadersWriter* and have it treat A as a read-style operation. As in the previous two exercises, this necessitates that operation A be re-implemented.
- Totally re-implement B, in a way that changes it into a read-style operation. This will necessitate a change in the synchronisation code. (This tests cell(3a, 3)).
- Introduce a new write-style operation, D. (This tests cell(3c, 3)).

The resulting SPC of the subclass is:

ReadersWriter[{A, B}, {C, D}]

The code for the derived class is shown in Figure 12.38 (the base class is as previously shown in Figure 12.36). As one might expect, since changes are made to both the sequential code and the synchronisation code, there is not much reuse. This exercise does not test cell(3b, 3). This can be remedied by changing the exercise so that operation A is incrementally modified instead of being re-implemented anew.

12.9.4 Discussion

Recall that there are two parts to the ISVIS conflict:

(i) Changes to the sequential code of a class might necessitate change to its synchronisation code.

(ii) Similarly, changes to the synchronisation code of a class might necessitate change to its sequential code.

The use of generic synchronisation policies does not *eliminate* part (i) of the ISVIS conflict. However, it does mean that whatever changes have to be made to synchronisation code can be achieved in just a single line of code (assuming that the policy instantiated in the

```
class Derived inherits Base {
  A(...) { ...}
  B(...) { ...}
  D(...) { ...}
synchronisation
  ReadersPriority[ {A, B},   {C, D} ]
  // alternatively...
  // ReadersPriority[ {A, B},   super.Ops − {A, B} + {D} ]
```

Figure 12.38: 9th Programming exercise (generic synchronisation policy)

subclass has already been written; if not then there will be extra work involved in writing this policy). Thus, although this part of the ISVIS conflict has not been eliminated, its harmful effects have been reduced to a trivial level. In particular, the (re-)instantiation of a generic synchronisation policy in a subclass requires just a single line of code irrespective of whether the instantiation is written anew or as an incremental modification of the instantiation of the policy in the parent class. This is always less (often *significantly* less) lines of code than is required to incrementally modify synchronisation code in other inheritance mechanisms surveyed. It is also less mental effort since generic synchronisation policies provide a higher level of abstraction than, say, guards or enabled-sets. As such, we claim that generic synchronisation policies represent a better way to tackle this part of the ISVIS conflict than the various techniques for inheriting and incrementally modifying synchronisation code that the other approaches advocate.

It is not surprising that the concept of GSPs is so effective at reducing the harmful effects of the ISVIS conflict. After all, the ISVIS arises out of two different uses of inheritance conflicting with one another. GSPs simply replace one of these uses of inheritance with genericity as the primary means to reuse code. By reducing the role of inheritance, it naturally reduces the harmful effects of the ISVIS conflict.

Part (ii) of the ISVIS conflict is often overlooked. The use of generic synchronisation policies does not help in this regard. If a change in synchronisation code necessitates changes in sequential code then those changes have to be made and the use of generic synchronisation policies will not be of any help.

12.10 Conclusions

In this chapter we have used the ISVIS matrix to evaluate how various approaches to the inheritance of synchronisation code conflict with the inheritance of sequential code.

None of the inheritance mechanisms *solve* the ISVIS conflict. (It is probably impossible to do so.) However, all the mechanisms surveyed use techniques to try to *reduce* its harmful

effects. These techniques generally fall into two categories: (i) keep synchronisation code and sequential code separated from one another so that the inheritance of one is syntactically separate from the inheritance of the other; and (ii) provide a means to incrementally modify inherited synchronisation code. Many but not all mechanisms adopt both approaches. For example, Eiffel|| employs the first approach but not the second. Conversely, Rosette and Arche use the second approach but not the first. The most effective inheritance mechanisms seem to be those that employ both techniques.

We propose an alternative approach: that genericity, rather than inheritance, be used as the primary means of reusing synchronisation code. We have shown in Section 12.9 that the use of generic synchronisation policies dramatically reduces the harmful effects of the ISVIS conflict. It is both shorter and easier to re-instantiate a generic policy than it is to try to incrementally modify inherited synchronisation code.

Chapter 13

Related Work and Summary of Contributions

This chapter brings Part IV of the thesis to a close. We start in Section 13.1 with a review of previous research into the ISVIS conflict. Then in Section 13.2 we summarise our own contributions in this area and finish off in Section 13.3 with some suggestions for future work.

13.1 Related Work

This section presents a historical overview of previous research into (what we call) the ISVIS conflict. This overview shows that, to date, there has been an almost universal misunderstanding within the research community regarding the nature of the conflict.

13.1.1 Early Research into the Problems with the Use of Inheritance in COOPLs

It has been known since at least 1987 that there are problems with the use of inheritance in COOPLs. For example, America [Ame87] noted that in the POOL language, inheritance could be employed to reuse the sequential methods of a class but it did not seem practical to inherit the "body" (akin to the *Live* routine in Eiffel||) of the class.

Awareness of the problem increased in 1989 with the publication of two papers, both of which showed that the problem with inheritance might hinder not just the reuse of inherited synchronisation code but also the reuse of inherited sequential code. The message in both papers was effectively that "inheritance and [synchronisation] tend to interfere with each other" [KL89, pg. 297] [TS89, pg. 103]. This wording implied that it might be possible to tackle the problem by developing new synchronisation mechanisms that do *not* conflict/interfere with inheritance. In the first of these papers [KL89], Kafura and Lee proposed the *behaviour abstraction* mechanism for the ACT++ language. This was slightly modified by Tomlinson and Singh [TS89] for the Rosette language and the mechanism was renamed as *Enabled-sets*.

Behaviour abstraction and Enabled-sets were presented as being synchronisation mechanisms. However, it would have been more apt to have described them as being inheritance (of synchronisation code) mechanisms. For example, the synchronisation construct in Enabled-sets was the **become** statement which caused a change in synchronisation state and in so doing determined what operations could execute next. Similar synchronisation capabilities had already existed, albeit in slightly different form, in Hybrid [Nie87]. What set behaviour abstraction and Enabled-sets apart from previous mechanisms was that the operations permitted to execute next were not specified *directly* as in Hybrid but rather *indirectly* via a set that could be inherited and incrementally modified in a subclass. It was this *inheritance* capability, rather than the synchronisation mechanism *per se*, that reduced some of the harmful effects of the ISVIS conflict.

This presentation of an inheritance (of synchronisation code) mechanism as a synchronisation mechanism marked the start of what has become a long-standing confusion over the nature of the ISVIS conflict and, consequently, how the conflict might be tackled.

Other, early papers also indicated that the problem was due to an interference between synchronisation and inheritance and claimed to have developed synchronisation mechanisms that were integrated with inheritance [Neu91] [BI92] [GW91, pg. 193].

13.1.2 Matsuoka's Analysis of the Inheritance Anomaly

Matsuoka realised that the mechanisms proposed in Act++ and Rosette did not provide general solutions to the problems associated with the use of inheritance in COOPLs. He performed his own analysis and developed a suite of programming exercises to illustrate other ways in which the *inheritance anomaly*, as he called it, might manifest itself [MWY90] [MY90] [MY93] [Mat93]. Unfortunately, Matsuoka did not see that the problem was caused by two uses of inheritance conflicting with one another. Rather, like others before him, he thought that at "the heart of the problem is the semantical conflicts between the descriptions of object-wise synchronisation and inheritance" [MY93, ps. 3] [Mat93, pg. 17].

This belief that the inheritance anomaly had its roots in synchronisation showed up in Matsuoka's classification of three categories of the inheritance anomaly. This classification was given in terms of *states*, e.g., as used in Enabled-sets:

- **Partitioning of acceptable states.**
  This occurred if a subclass split one of the inherited states in two. For example, if a subclass of a bounded buffer introduced a new operation, *Get2*, that removed two items from the buffer then this would result in the *partial* state being split into two states: *one-item* and *two-or-more-items*.
- **History-only sensitiveness of acceptable states.** This was illustrated by a subclass that introduced a new operation, *GGet*, that was similar to *Get* except that it could not execute immediately after an invocation of *Put*. This new operation was said to be "history-sensitive" because its synchronisation constraints depended on the history of

prior invocations upon the object.

- **Modification of acceptable states.** This occurred if a subclass made a change that modified the states (conditions) under which most or all of the inherited operation could execute. The example given was that of a subclass of a bounded buffer that introduced two new operations, *lock* and *unlock*. The inherited operations, *Put* and *Get*, could execute only if the object was unlocked; hence the states (conditions) under which they could execute had been modified.

There are several points to note about this classification.

Firstly, the "state" mentioned in each of these three categories is *synchronisation* state and thus the names of Matsuoka's three categories imply synchronisation (rather than inheritance) as being the cause of the inheritance anomaly.

Secondly, we discussed in Section 11.5.2 why one should ensure that the programming exercises used to evaluate an inheritance (of synchronisation code) mechanism do not require more expressive power than that available in the associated synchronisation mechanism. To recap, failure to ensure this might lead to programmers having to mix synchronisation code with sequential code in order to implement the programming exercises. This mixing, rather than any inherent limitation of the inheritance mechanism, can hinder code reuse.

The programming exercises Matsuoka used for the history-only sensitivity and modification of acceptable state categories required that a subclass maintain extra variables. For example, synchronisation of the *GGet* operation required that a boolean variable, *after_put*, be maintained. Similarly, preventing *Put* and *Get* from executing if the buffer was locked required the maintenance of a boolean variable, *is_locked*. These variables were synchronisation variables. However, since many synchronisation mechanisms surveyed by Matsuoka did not have sufficient expressive power to maintain synchronisation variables, they were maintained as instance variables in the subclass. In effect, Matsuoka used programming exercises that demanded high expressive power and hence were unsuitable for the evaluation of inheritance (of synchronisation code) mechanisms.

One result of this was that Matsuoka noticed that different synchronisation mechanisms avoided different categories of the inheritance anomaly. (This was because synchronisation mechanisms varied in the expressive power they possessed, but Matsuoka did not comprehend this.) He therefore proposed that a language support *several* synchronisation mechanisms, reasoning there was a good chance that, between them all, these mechanisms might avoid most/all categories of the inheritance anomaly [MTY93, pg. 113]. For example, if the use of a particular mechanism to implement the synchronisation policy for a base class resulted in subclasses suffering a particular category of inheritance anomaly then a programmer could re-implement the synchronisation code of the base class in a different mechanism that did not result in subclasses suffering that category of inheritance anomaly.

Although Matsuoka claimed that there were three categories of inheritance anomaly, he never actually defined what an inheritance anomaly was supposed to be. This lack of a clear

definition was probably due to the fact that Matsuoka tried to analyse and categorise the inheritance anomaly in terms of synchronisation, which had nothing to do with the problem.

13.1.2.1 The Spread of the Misunderstanding

Matsuoka's analysis was widely disseminated within the research community, and was highly praised as being "pioneering work" [Ber94, pg. 113] and "an excellent analysis and survey of the anomaly" [Mes93, pg. 221]. With it was spread the basic misunderstanding of the nature of the problem, i.e., that it was a conflict between synchronisation and inheritance which might be solved by developing new synchronisation mechanisms that did not conflict with inheritance.

Soon researchers started developing synchronisation/inheritance mechanisms that implemented all of programming exercises used in Matsuoka's analysis and they claimed that they had solved the inheritance anomaly [Mes93] [Löh93] [LL94] [Tho94]. From reading these papers, one would be lead to believe that the "solutions" were due to the synchronisation mechanisms. This was partly true as the mechanisms had enough expressive power to handle Matsuoka's exercises. However, as with previous work, it was rather that these mechanisms provided a means to *inherit* synchronisation code that lead to a reduction of the harmful effects of the ISVIS conflict.

13.1.3 Meseguer's Attempt to Eliminate Synchronisation Code

As our analysis in Chapter 11 shows, the ISVIS conflict is due the inheritance of synchronisation code conflicting with the inheritance of sequential code. Thus, it seems logical that the conflict could be resolved by eliminating one of these forms of inheritance. This is, in effect, what the use of GSPs does: it eliminates the inheritance of synchronisation code and replaces it with genericity. In so doing, it considerably reduces (though does not entirely eliminate) the harmful effects of the ISVIS conflict.

However, if a person misunderstood the nature of the problem and thought that the conflict was between synchronisation and inheritance then they might reason that the conflict could be resolved by eliminating synchronisation. Such an approach was taken by Meseguer who claimed that:

> "The inheritance anomaly [is] a problem *caused* by the very presence of synchronisation code. The logical solution if we take this hypothesis seriously is to *completely eliminate* synchronisation code" [Mes93, pg. 221] (emphasis in the original).

Despite the incorrect assumption that the cause of the inheritance anomaly is rooted in synchronisation, this approach had some validity since, by eliminating synchronisation code, there could be no conflict between the *inheritance* of synchronisation code and the *inheritance* of sequential code.

However, Meseguer did not "completely eliminate" synchronisation code as he claimed, but rather made synchronisation code *implicit* in the declarative coding style of the Maude language [Mes93]. Unfortunately, this did not work to eliminate the inheritance anomaly, as we now discuss.

One example in the paper on Maude [Mes93, pg. 234] involved a subclass of a bounded buffer that introduced a new operation, *GGet*. This operation was similar to *Get* except that it could not execute immediately after *Put*. In order to implement this, a boolean variable, *after_put*, was introduced. This variable was claimed to be an instance variable but was actually a synchronisation variable that just happened to be *implemented* as an instance variable. Thus, rather than eliminating synchronisation code, Maude simply mixed it back in with sequential code.

In this particular example, *all* the inherited operations had to be re-implemented anew in order to maintain *after_put*. Meseguer claimed that the inability to reuse *any* inherited code in this example was not an inheritance anomaly and was, in fact, entirely appropriate. This was a rather surprising claim. Equally surprising was that other researchers [LL94] [Mat93] [Tho94] [Löh93] were apparently prepared to accept this claim at face value. This, we feel, is a result of trying to solve an ill-defined problem (the inheritance anomaly).

13.1.4 Bergmans' Analysis of the Inheritance Anomaly

As far as we know, there have been only two attempts, other than our own, to carry out a detailed analysis of the inheritance anomaly. The first, by Matsuoka, has already been discussed. In this section we discuss the other analysis, by Bergmans [Ber94].

Bergmans made the common mistake of thinking that the problem was due to "conflicts between inheritance and synchronisation" [Ber94, pg. 100]. Like Matsuoka, Bergmans claimed to know the causes of the inheritance anomaly, but never actually defined what an inheritance anomaly was supposed to be. According to Bergmans, the origins of the inheritance anomaly were:

- **Synchronisation modularity.**
  Synchronisation code should be syntactically separate from sequential code so that they can be inherited separately from one another.
- **Synchronisation granularity.**
  It is easier to inherit synchronisation code if it is written in several parts (e.g., as in Enabled-sets or guard-based mechanisms) rather than as one monolithic unit (e.g., as in Path Expressions or the *Live* routine in Eiffel||).
- **Expressiveness for synchronisation conditions.**
  If a synchronisation mechanism does not have sufficient expressive power to directly implement the synchronisation policy used in a class then programmers may have to resort to mixing synchronisation code with sequential code in order to implement the

policy. This, of course, results in a loss of modularity and hence may hinder the inheritance of code.

In the above list, the **bold** text is the actual titles that Bergmans gave to what he claimed were the causes of the inheritance anomaly [Ber94, pg. 100]. These titles clearly imply that the causes of the inheritance anomaly were rooted in synchronisation rather than in inheritance. However, we have shown in Chapter 11 that the ISVIS conflict is due to two uses of inheritance conflicting with each other. The "causes" that Bergmans claimed are not the actual origins of the conflict but rather are issues than can exacerbate the conflict. Even with full synchronisation modularity, suitably small synchronisation granularity and excellent expressive power, the ISVIS conflict will still exist.

13.1.5 Summary of Related Work

Virtually all the previous work on the problem of using inheritance in COOPLs has misunderstood the nature of the problem, thinking that it is a conflict between synchronisation and inheritance, rather than a problem rooted in inheritance. (It must be said that, for a time, the present author also misunderstood the nature of the conflict. An early version of the analysis of the ISVIS conflict in this thesis viewed the conflict as being between synchronisation and inheritance [MWBD92].) This basic misunderstanding has hindered progress in tackling the problem in two ways.

Firstly, this misunderstanding has prevented researchers from being able to actually define the problem, which in turn makes it rather difficult to solve the problem.

Secondly, researchers have thought that a solution to the problem lay in either developing new synchronisation mechanisms or eliminating synchronisation code entirely, rather than developing either new inheritance mechanisms or alternatives to inheritance. Many of the proposed "solutions" actually combine a new synchronisation mechanism with an inheritance mechanism but usually the synchronisation mechanism is emphasised instead of the inheritance mechanism as being the relevant factor in reducing the amount of code that has to be redefined in subclasses.

13.2 Summary of our Contributions

There are problems associated with the use of inheritance in COOPLs. It is widely thought that one of these problems is due to a conflict between synchronisation and inheritance [TS89] [KL89] [Neu91] [GW91] [Tho94] [Löh93] [BBI+] [Mes93] [BFS93] [Mat93] [Cou94] [Ber94]. Our analysis of this problem shows that this understanding of the nature of the problem is incorrect and that the problem is rooted in inheritance. In particular, the problem is due to a conflict between two uses of inheritance: (i) inheritance employed to reuse synchronisation code, and (ii) inheritance employed to reuse sequential code.

We have developed a tool, the ISVIS matrix, that can be used to examine the harmful

effects of the conflict when an inheritance mechanism for sequential code interacts with an inheritance mechanism for synchronisation code. Our survey of how a common form of single-inheritance for sequential code interacts with several inheritance mechanisms for synchronisation code confirms that the harmful effects of the conflict can be reduced if (i) synchronisation code and sequential code are separated from one another, and (ii) a means is provided for a subclass to incrementally modify inherited synchronisation code. However, our survey also shows that incremental modification of synchronisation code has two limitations. Firstly, it can be verbose, often requiring as much code as re-implementing the desired synchronisation policy anew. Secondly, it cannot always be used, thus *necessitating* that synchronisation code be rewritten anew.

The approach we propose to reduce the harmful effects of the ISVIS conflict is to employ generic synchronisation policies. Since a subclass can (re-)instantiate a generic synchronisation policy in just a single line of code, this approach is often considerably shorter than trying to incrementally modify inherited synchronisation code. Also, since generic synchronisation policies are at a high level of abstraction, it usually less error-prone to re-instantiate a generic synchronisation policy than to incrementally modify, say, inherited guards.

13.3 Limitations of our Research and Future Work

We analysed the harmful effects of the ISVIS conflict for a common form of single-inheritance for sequential code. However, as we pointed out in Section 11.5.1, other forms of inheritance for sequential code also exist. These should also be examined to see if some inheritance constructs are more (or less) likely to result in harmful ISVIS conflicts than other inheritance constructs.

Aside from the ISVIS conflict, there is another problem with the use of inheritance in COOPLs: it can violate encapsulation (as was briefly discussed in Section 11.1. This is another area that warrants study.

Finally, we have shown that the ISVIS conflict is not peculiar to synchronisation but rather is intrinsic to inheritance. This finding suggests that similar conflicts may arise in other fields where programmers try to inherit several different kinds of code in a single class hierarchy. For example, problems in the area of real-time programming have been noted by Askit *et al.* [ABvdSB94]. However, the language and analysis they use is quite similar to that of researchers who claim that the inheritance anomaly is a conflict between synchronisation and inheritance. This clearly indicates two things. Firstly, that the conflicts between the inheritance of different kinds of code are not confined to the area of synchronisation. Secondly, that confusion over the nature of such conflicts is widespread in other areas too.

Part V

Conclusions

Introduction to Part V

Part V brings this thesis to a close. Chapter 14 outlines several possibilities for future work. Then Chapter 15 summarises the main contributions of this thesis.

Chapter 14

Future Work

This chapter suggests some possibilities for future work. Most of these possibilities are discussed via examples. For instance, Section 14.3 outlines a sample timeout mechanism in order to show that it is feasible to extend the Sos paradigm in this direction. In such cases, the sample mechanisms are presented purely for illustrative purposes and are not intended as polished designs. None of the sample mechanisms discussed in this chapter are implemented.

14.1 Summary of Future Work Discussed in the Body of this Thesis

We start this chapter by briefly summarising some areas for future work that have already been discussed at length in the main text of this thesis.

Synchronisation in client objects. The Sos paradigm concerns itself solely with synchronisation in *service* objects. Some languages provide support for synchronisation in *client* objects—futures [Hal85] and wait-by-necessity [Car93] are two examples of constructs for client object synchronisation. Of course, there is no strict division between client and service objects since it is common for a service object to be, itself, a client of other objects. As such, if a language provides constructs for both client object synchronisation and service object synchronisation then it is important that these constructs work together in a harmonious fashion.

Further analysis of the ISVIS conflict. Our analysis of the ISVIS conflict is incomplete in two ways.

Firstly, our examination of the inheritance of sequential code was confined to a common form of single inheritance. As we mentioned in Section 11.5.1, other forms of inheritance also exist: multiple inheritance, Beta's "inner" construct, Mixin-based inheritance, the ability of a subclass to "undefine" inherited operations and so on.

Secondly, our examination of the inheritance of synchronisation code was confined to inheritance of synchronisation in *service* objects. It is possible that the ISVIS conflict also exists for the inheritance of synchronisation constructs, e.g., futures [Hal85], in *client* objects.

Obviously, further research is required to obtain a more complete understanding of the ISVIS conflict.

Optimisation of Esp. This thesis has briefly mentioned two techniques for optimising Esp: the re-evaluation matrix and optimisation by transformation. However, we have not invested much time to develop and apply these techniques. Optimisation of Esp—whether it be performed by these or other techniques—is of great importance because, as our prototype implementation has shown, a naive implementation of Esp can impose an overhead of hundreds of instructions per invocation of a synchronised operation. This overhead limits the use of Esp to applications that utilise only coarse-grained concurrency.

Open issues for generic synchronisation policies. Chapter 9 discussed some open issues regarding generic synchronisation policies.

Firstly, generic synchronisation policies fully encapsulate the implementation details of a policy. Even the synchronisation mechanism used to implement a policy is hidden from programmers who instantiate the policy. One possibility this raises is that a language might support several synchronisation mechanisms. Another possibility is that generic synchronisation policies might act as "glue" to interface between general-purpose programming languages and special-purpose languages in which synchronisation policies are written.

Secondly, our prototype implementation of Gasp permits a subclass to either inherit the instantiation of a generic synchronisation policy "as is" from its parent class or to re-instantiate it anew. The examples used in Section 12.9 indicate that it would be useful for a subclass to be able to incrementally modify the instantiation of an inherited policy. However, our prototype implementation does not support this. We do not foresee any difficulty in providing this capability. However, as discussed in Section 11.1, this capability ties in with the issue of inheritance violating encapsulation. As such, we feel that a detailed analysis of the conflict between inheritance and encapsulation should be undertaken before providing such support.

Finally, it would be useful to be able to instantiate several generic synchronisation policies upon the operations of a class. However, as discussed in Section 9.3.2, this raises several thorny issues which need to be addressed if such support is to be provided.

14.2 Development of Other Sos-based Synchronisation Mechanisms

This thesis has used just one synchronisation mechanism, Esp, to illustrate the concepts of the Sos paradigm. However, Sos is not limited to Esp and the concepts of the Sos paradigm

could be applied to develop other mechanisms.

As an example to show just how different other SOS-based synchronisation mechanisms might be from ESP, this section presents a brief overview of another mechanism. Note that this mechanism is just a paper design and has not been implemented. We introduce this synchronisation mechanism, which is queue-based, through some examples. Then afterwards we discuss some of its important points and compare it to ESP.

First-come, First-served Printer

The code in Figure 14.1 implements a FCFS printer. A single queue, *PrintQ*, is declared. This is sorted by the relative arrival time of pending invocations. Actions are used to insert *Print* invocations into this queue at their *arrival* and remove them from the queue when they *start* execution. (Unlike ESP, programmers are responsible for maintaining queues of invocations.) The guard for the invocation at the head of the queue expresses mutual exclusion but it does not include anything to specify scheduling order since that is implicit in the maintenance of the queue.

```
class FCFSPrinter {
  Print(...) { ... }
synchronisation
  queue  PrintQ sorted by arr_time;

  arrival(Print) → { PrintQ.insert(this_inv); }
  start(Print) → { PrintQ.remove_head(); }
  PrintQ.head_guard: exec(Print) = 0;
}
```

Figure 14.1: FCFS policy expressed in terms of a queue

Starvation-free, Shortest Job Next Scheduler

The code in Figure 14.2 implements a starvation-free version of the Shortest Job Next scheduler. There are just two differences between the code for this policy and the code for the FCFS printer (Figure 14.1). The first is the different sorting order of *PrintQ*. The second is the *start*(Print) action. The latter iterates over invocations in *PrintQ* and decrements the *length* of any that were skipped over. Since doing so may leave *PrintQ* in an unsorted state, it is explicitly re-sorted afterwards.

Alarm Clock

The code in Figure 14.3 implements an alarm clock. As in the ESP implementation of this problem (Figure 4.5 on page 51), we assume that the run-time system will invoke *Tick*

```
class FairSJN {
  Print(int length, ...) { ... }
synchronisation
  queue  PrintQ sorted by length, arr_time;

  arrival(Print) → { PrintQ.insert(this_inv); }
  start(Print) →
  { PrintQ.remove_head();
    for p in PrintQ do
      if p.arr_time < this_inv.arr_time then p.length --; endif
    end
    PrintQ.sort();
  }
  PrintQ.head_guard: exec(Print) = 0;
}
```

Figure 14.2: Starvation-free, Shortest Job Next scheduler

periodically to mark the passage of time.

The synchronisation code maintains two queues: one for *WakeUp* and another for *Tick*. Since the guard for invocations of *Tick* is **true**, invocations will be removed from *TickQ* almost as soon as they are inserted into it. In this case it may seem rather annoying to have to maintain a queue for *Tick* at all, but that is the nature of this synchronisation mechanism. At a *WakeUp* invocation's *arrival*, its *wakeup_time* (a synchronisation local variable) is calculated and this is used to sort *WakeUpQ*. This sorting order means that the *WakeUp* invocation to be woken up next is at the head of the queue. The resultant guard for the head element of *WakeUpQ* is trivial and intuitive.

14.2.1 Discussion

It is easy to see that this synchronisation mechanism is SOS-based since it embraces the four concepts of SOS: (i) the mechanism is event-based and permits actions to be associated with them; (ii) it employs a construct, guards, to cause pending invocations to start executing; (iii) synchronisation code and variables are separated from sequential code and variables; and (iv) the mechanism permits synchronisation code to access information about invocations.

There are two main difference between this mechanism and ESP.

The first difference concerns the management of invocations. ESP automatically maintains collections of *waiting* and *executing* invocations, while this mechanism requires programmers to explicitly maintain queues of pending invocations themselves. Thus in this mechanism programmers have to do more "housekeeping" work.

The second difference concerns guards. Although both mechanisms employ guards, this

```
class AlarmClock {
  WakeUp(int period) { }
  Tick() { }
synchronisation
  int   wakeup_time local to WakeUp;
  queue   WakeUpQ sorted by wakeup_time;
  queue   TickQ sorted by arr_time;

  arrival(Tick) → { TickQ.insert(this_inv); }
  start(WakeUp) → { TickQ.remove_head(); }
  TickQ.head_guard: true;

  arrival(WakeUp) →
  { this_inv.wakeup_time := term(Tick) + this_inv.period;
    WakeUpQ.insert(this_inv);
  }
  start(WakeUp) → { WakeUpQ.remove_head(); }
  WakeUpQ.head_guard: term(Tick) >= this_inv.wakeup_time;
}
```

Figure 14.3: Implementation of the Alarm Clock problem via queues

mechanism evaluates guards *only* for invocations at the head of queues while Esp potentially evaluates guards for *all* pending invocations at each event.

These differences are enough to result in the two mechanisms having quite a different "feel" to one another.

One advantage of this mechanism over Esp is that an unoptimised implementation of it will be significantly more efficient than an unoptimised implementation of Esp. This is because in re-evaluating the guards *only* of invocations at the head of queues rather than the guards of *all* pending invocations, it cuts down dramatically on guard re-evaluation.

A disadvantage of this mechanism is that while its explicit support for queues makes it very suitable for scheduling/queuing policies, it does not lend itself to some other types of synchronisation policies. For example, while this mechanism can implement, say, the Dining Philosophers problem, it cannot do it in an elegant manner. (The only implementation approach that comes to mind is one which requires that a separate queue be maintained for each table position.) As such its expressive power is not as good as that of Esp.

However, the most important point of this mechanism is that it is proof that Esp is not the only synchronisation mechanism that can be designed within the Sos paradigm. It will be interesting to see how Sos might be used in the future to develop other mechanisms.

14.3 Support for Timeouts in the Sos Paradigm

A *timeout* is a period of time within which a pending invocation must *start* being serviced.[1] If the invocation is delayed for longer than this time before it can be serviced then the invocation is aborted and an exception raised in the caller. Timeouts are provided in some languages as a way to support real-time programming.

In this section we show by example how the Sos paradigm might be extended to provide support for timeouts. All that is required is the introduction of two new events—*timing* and *timeout*—and an extra synchronisation variable local to invocations: *timeout_value*.

As its name suggests, *timeout_value* indicates the timeout value of an invocation. The default value is, say, **infinite**, indicating that invocations do not timeout. This default can be overridden by a client when making an invocation. Example syntax to illustrate this is shown below:

```
obj.foo(...);      // infinite timeout by default
obj.foo(...) with timeout_value = 1.2 seconds;
```

Alternatively, synchronisation code in the service object might set a timeout value for invocations. For example, the following action assigns a *timeout_value* for invocations that have the default value while honouring the *timeout_value* of invocations whose clients have explicitly set it.

```
arrival(Foo) →
{ if this_inv.timeout_value = infinite then
    this_inv.timeout_value := 10 seconds;
  endif
}
```

For *timeout_value* to be useful, the run-time system needs to automatically generate *timing* events to denote the passage of time. At these events, the *timeout_values* of *waiting* invocations are examined and if an invocation has timed out then a *timeout* event is generated for it. (Note that it is unnecessary, and indeed wasteful, for *timing* events to be generated periodically. Rather the run-time system might arrange for *timing* events to be generated only when invocations are due to timeout.) It is probably most convenient if the run-time system handles *timing* events automatically so that programmers need not concern themselves with them. In fact, programmers need not even be aware that they exist. On the other hand, *timeout* events are potentially of interest to programmers. For example, just as the run-time system will, say, decrement the relevant *wait* synchronisation counter whenever an invocation times out, so too might programmer-maintained synchronisation variables need to

[1] Some languages might define a timeout to be a period of time within which a pending invocation must *terminate* (rather than *start*) execution. It is even possible that a language might support *both* forms. Whichever way it is defined is immaterial to the present discussion since the principles are the same.

be updated; for this purpose programmers need to be able to associate actions with *timeout* events.

The proposal outlined in this section is very tentative. However, it does show that adding support for timeouts in SOS is feasible.

14.4 Exception Handling

Support for exceptions is becoming more and more common in object-oriented languages. The form that such support takes varies dramatically from one language to another and, as such, it can be difficult to generalise about exception mechanisms. However, one thing that can be said is that languages that support exceptions provide ways to (i) *raise* an exception, and (ii) *handle* an exception. We discuss exception handling here in Section 14.4, and exception raising in Section 14.5.

In this section we discuss why it is important that a synchronisation mechanism be integrated with the host language's exception *handling* mechanism. We then briefly outline a way in which such integration might be achieved for SOS-based synchronisation mechanisms. Section 14.5 discusses similar issues as they apply to a particular exception *raising* mechanism; that of the Eiffel language.

14.4.1 The Need to Integrate Synchronisation with Exception Handling

An exception handler is a segment of code that will be executed when an exception is raised during execution of an operation. The task of the handler is to attempt to process the exception gracefully. In doing so it might be able to resume execution of the operation. If not then the handler should place the variables of the object in a consistent state and re-raise the exception in the caller. In this case, execution of the operation in which the exception occurred is terminated abnormally. The possibility for abnormal termination has consequences for synchronisation code as we now explain.

The SOS paradigm models the lifetime of an invocation in terms of three events: *arrival*, *start* and *term*. The last of these, *term*, denotes the *normal* termination of an invocation. SOS does not have any event that denotes an *abnormal* termination of an invocation, i.e., via an exception. What happens if an operation is terminated abnormally is implementation dependent. One possibility is that the abnormal termination will be treated by the synchronisation code as a normal *term* event. Another possibility is that the synchronisation code will not be made aware of the abnormal termination at all—this is the case in the DESP and GASP compilers. In this case, the synchronisation code cannot take steps to ensure that synchronisation variables are left in a consistent state when an invocation terminates abnormally. For example, the *exec* counter would not be decremented and information about the aborted invocation would not be removed from *executing* list. Clearly, this is a problem. Strangely, it is a problem that has gone unnoticed in the literature and we are not aware of

any synchronisation mechanisms for object-oriented languages that explicitly take steps to ensure compatibility with exceptions.

14.4.2 A Proposal

One of the tasks of a *sequential exception handler* is to place the sequential (instance) variables of an object in a consistent state when execution of an operation is terminated abnormally. In order to integrate a synchronisation mechanism with exception handling, it is necessary to introduce the complimentary concept of a *synchronisation exception handler*. This would have the task of placing the synchronisation variables of an object in a consistent state when execution of an operation is terminated abnormally.

The obvious way in which to achieve this in the Sos paradigm is to introduce a new event, *exception*, which will occur if the execution of an operation is terminated abnormally. As with the other, existing events—*arrival*, *start* and *term*—the run-time system might execute some code automatically at this event, e.g., decrement the *exec* counter and remove information about the aborted invocation from the *executing* list. Programmer-specified actions at *exception* events would be used to ensure the consistency of programmer-maintained synchronisation variables. Presumably, the introduction of the *exception* event would be complimented by the introduction of a new synchronisation variable local to invocations, *exception_name*, to indicate the reason for the exception.

14.5 Exception Raising via Eiffel-style Assertions

The exception raising mechanism of the Eiffel language [Mey92] is based on assertions. If an assertion evaluates to false then an exception is raised. Eiffel's assertions can take a number of forms. For example, programmers can specify variants and invariants in a looping construct. Also assertions can be placed anywhere in the body of an operation via a **check** statement. However, of more interest to this discussion are the forms of assertions that can be used *outside* of the bodies of operations. There are three such forms: *preconditions*, *postconditions* and *class invariants*. Pre- and postconditions are associated with operations and are evaluated before and after execution of the operation, respectively. If the post-conditions of all the operations of the class contain a common component then this component can be factored out into what is termed the *class invariant*. This is often done as an aid to code readability.

From here on, the term *assertions* is used to refer only to those assertions that appear outside of the bodies of operations, i.e., pre- and postconditions, and class invariants.

14.5.1 Incompatibility of Eiffel-style Assertions and Internal Concurrency

Eiffel's assertions are expressed in terms of the instance variables of an object. (Pre- and post-conditions may also access parameters but that is of no relevance to the present discussion.) Thus one may consider Eiffel's assertions to be *readers* of an object's instance variables. Like-

wise, operations that update instance variables can be considered to be *writers*. If a language supports both concurrency within objects and Eiffel-style assertions then this introduces the age-old readers/writer problem. This is because an operation that updates instance variables might be executing while, say, the pre-condition of another operation is being evaluated.

Simply put, Eiffel-style assertions are incompatible with concurrency within objects. This problem has also been noted by Benveniste and Issarny [BI92, pg. 11]. At least one language, CEiffel [Löh93], dangerously supports both Eiffel-style assertions and concurrency within objects. Several other concurrent extensions to Eiffel keep its assertion mechanism but prohibit concurrency within objects [KB93] [Mey93] [Car93] thus avoiding the problem. However, it is not clear if the prohibition against internal concurrency in these proposals was made as a result of an awareness of the incompatibility or for other reasons.

14.5.2 Incompatibility of Eiffel-style Assertions and the Sos Paradigm

When viewed in the context of the Sos paradigm, other incompatibilities between Eiffel-style assertions and internal concurrency become apparent. These problems centre around *when* assertions are evaluated.

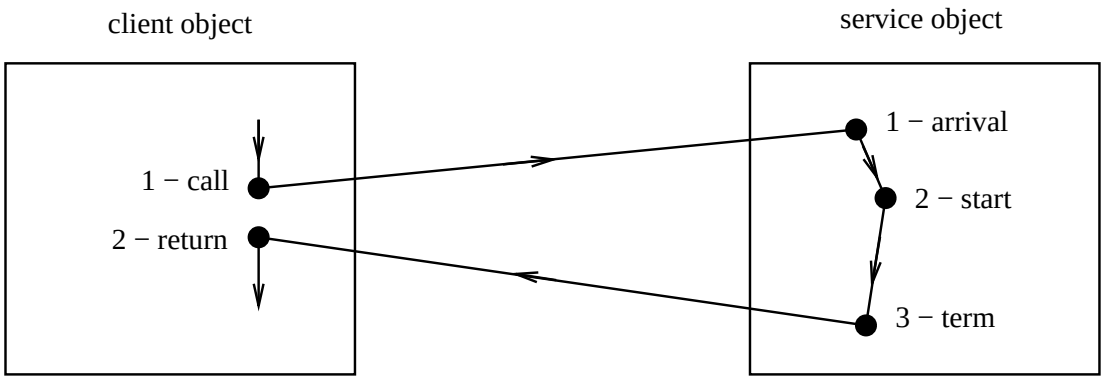

Figure 14.4: The events in the lifespan of a typical operation invocation

Consider the diagram in Figure 14.4. In a sequential language, the events *call*, *arrival* and *start* are all merged together and it does not make sense to distinguish at which one of these events the precondition of an operation is evaluated.[2] However, in a concurrent system these events are separate and it is necessary to define exactly when a precondition is evaluated. Intuitively, a precondition is an assertion that must be true when an operation starts execution. As such it seems that it must be evaluated at the *start* event. If it were evaluated sooner—at either the *call* or *arrival* events—then there is no guarantee that it would still hold true at the *start* event since in the meantime another concurrently executing operation might have updated some instance variables that are used in the precondition.

[2] For example, Eiffel does not define whether the pre- and post-conditions of an operation are evaluated by the caller or the callee. However, for practical reasons, most, if not all, implementations of Eiffel evaluate pre- and post-conditions in callee, i.e., in the body of the operation being invoked.

Likewise, a sequential language does not distinguish between the *term* and *return* events for the evaluation of a postcondition. Intuitively, a postcondition is an assertion that must be true when an operation terminates execution so it should be evaluated at the *term* event. If it were evaluated later, at the *return* event, then there is no guarantee that it would still hold true due to the possibility of another concurrently executing operation having updated some instance variables used in the postcondition.

Unfortunately, there is a problem. If pre- and postconditions are to be evaluated *at* events then they must be evaluated by synchronisation code which is executed in a critical region. This can lead to deadlock if the assertions access instance variables indirectly by invoking, say, boolean operations on the object, e.g, *IsFull* or *IsEmpty* for a bounded buffer. This is because such operations might be synchronised which would result in the code trying to enter the critical region recursively. To avoid this, the compiler would have to arrange for unsynchronised versions of operations used in assertions to be invoked when evaluating assertions. This, of course, leads to an inconsistency in the semantics of operations. An operation, say, *IsFull* will be synchronised if invoked directly in an application, but will be unsynchronised if invoked as part of an assertion evaluation.

Some readers may think that this problem can be solved by evaluating an operation's pre-condition not *at* the *start* event but just *after* it, as the first statement inside the body of the operation; and likewise an operation's postcondition might be evaluated as the last statement inside its body, i.e., just *before* the *term* event rather than *at* it. However, this would still be prone to deadlock. To see why, consider a bounded buffer that includes operations *Put* and *IsFull*. Since *Put* updates the buffer's state and *IsFull* examines it, these two operations will probably execute in mutual exclusion with respect to each other. If the precondition of *Put* includes the expression "**not** IsFull()" and this precondition is evaluated *inside* the body of *Put* then deadlock will result since *IsFull* will be prevented from executing due to the fact that *Put* is already executing.

One way to solve these problems is to exploit the similarity between assertions and guards, as we discuss in Section 14.5.3.

14.5.3 Similarities between Assertions and Guards

Consider a bounded buffer with operations *Put* and *Get*. If the buffer is to be used in a sequential environment then suitable preconditions for these operations might be as follows:

Pre(Put): "the buffer is not full";
Pre(Get): "the buffer is not empty"

These are quite similar to the guards that might be used on a bounded buffer in a concurrent environment, e.g.:

Put: *exec*(Put) = 0 **and** "the buffer is not full";
Get: *exec*(Get) = 0 **and** "the buffer is not empty"

The synchronisation guards are similar to the preconditions, except that they contain extra constraints for the purpose of mutual exclusion. In a language that does not support concurrency within objects, these mutual exclusion constraints would not have to be specified and the preconditions would look identical to the synchronisation guards. This similarity between preconditions and guards used for synchronisation is well known. However, there is actually very little written about this issue in the literature.

There is a second similarity between preconditions and guards: the problem of safe access to instance variables in an object that supports internal concurrency. We explained earlier (in Section 14.5.1) that it is unsafe to evaluate the pre- or postcondition of an operation while instance variables it is accessing are being updated by another concurrently executing operation. This is quite similar to the problem of synchronisation code (e.g., in the form of guards) accessing instance variables while they are being updated by an operation.

The way the Sos paradigm solves the problem of synchronisation code having unsafe access to instance variables is to forbid such access and instead provide programmers with the ability to maintain their own synchronisation variables. This begs the question: can a similar approach be used to tackle the problem of assertions having unsafe access to instance variables in a concurrent environment? The answer is yes, as we now discuss.

14.5.4 A Proposal for an Event-based Assertion Mechanism

Informally, pre- and postconditions are evaluated at the *start* and *termination* of operations. This suggests that the Sos paradigm, which includes both *start* and *term* events, might provide a suitable infrastructure to support these assertions. Indeed, it would also permit an assertion to be evaluated at *arrival* events which, as we show later, can be useful.

However, the Sos paradigm permits actions to be executed at events. If both an action and an assertion are specified for a particular event then this raises the issue of whether the action should be executed *before* the assertion is evaluated, or *afterwards*. Both possible orderings make good semantic sense and there is no reason to support one instead of the other. As such, we suggest that both be supported. We use the notation *pre_x_assertion* to denote an assertion that is evaluated at event x (where x is one of *arrival*, *start* or *term*) *before* execution of the action for that event. Similarly, *post_x_assertion* denotes an assertion that is evaluated *after* execution of the action for that event. As an example to illustrate the syntax, the following indicates an assertion that is to be evaluated at the *start*(Put) event, before any action that might be specified for that event:

pre_start_assertion(Put): num < Size;

The ability to specify assertions for both before and after the execution of actions of three different event gives rises to potentially six different assertions in all. While this offers more flexibility than Eiffel's two (pre- and postconditions), it also offers the potential for more complexity.

We now present an event-based assertion mechanism in order to show that the SOS paradigm is powerful enough to express not just synchronisation constraints but also assertions. This mechanism is presented through examples. Note, however, that this mechanism is just a paper design and has not been implemented.

A Sequential Bounded Buffer with Assertions

We start by using the event-based assertion mechanism to specify constraints on a buffer to be used in a sequential environment; a later example will modify the buffer to take synchronisation constraints into account too. The class in Figure 14.5 is split in two by the keyword **assertions**. Preceding it is the sequential (functional) code of the class, while assertion-checking code follows this keyword. The assertions are expressed in an ESP-like syntax. The *pre_start_assertions* specify under what conditions the operations *Put* and *Get* can be invoked, and the *post_start_assertions* give guarantees about the effect of executing these operations. The assertion code declares a variable, *old_num*, local to invocations of *Put* and *Get*. This variable records the value of *num* when an invocation starts and is used by the assertion at the *term* event. This is akin to the usage of **old** in Eiffel postconditions [Mey92].

```
class Buffer[elem] {
  ...   // instance variables
  Buffer(int Size) { ... }   // constructor
  Put(...) { ... }
  Get(...) { ... }
assertions
  int   Size, num;
  int   old_num local to Put, Get;
  start(Buffer) → { Size := this_inv.Size; num := 0; }

  pre_start_assertion(Put): num < Size;
  start(Put) → { this_inv.old_num := num; }
  term(Put) → { num ++; }
  post_term_assertion(Put): num = this_inv.old_num + 1;

  pre_start_assertion(Get): num > 0;
  start(Get) → { this_inv.old_num := num; }
  term(Get) → { num --; }
  post_term_assertion(Get): num = this_inv.old_num - 1;
}
```

Figure 14.5: Event-based assertions for a sequential Bounded Buffer

There are several other points to note about this example.

Firstly, there is actually no need to explicitly maintain the variable *num* since it is equivalent to the expression "*term*(Put) – *term*(Get)". However, the code maintains it as an example to illustrate the usage of actions in this assertion mechanism.

Secondly, the assertions are expressed in a manner that is separate from, and independent of, the *implementation* of a class. This raises the interesting possibility of using an event-based assertion mechanism as part of a class's *type specification*.

Thirdly, the code appears to be more verbose than an equivalent class using Eiffel-style pre- and postconditions. This is because maintenance of assertion variables is clearly visible while, in Eiffel, pre- and postconditions are expressed in terms of instance variables, the maintenance of which is "hidden" in the bodies of the operations. We argued in Chapters 3 and 4 that in practice there is no overlap between instance variables and synchronisation variables; we do not know if the same holds true for the overlap of instance variables and assertion variables (and this is certainly an issue that needs to be studied). If not then programmers may frequently have to maintain pairs of variables in step: one an instance variable and the other an assertion variable.

Finally, due to the similarity of preconditions and synchronisation guards, there is bound to be some overlap between assertion variables and synchronisation variables. For example, the bounded buffer example (Figure 14.5) declares three assertion variables: *Size*, *num* and *old_num*. The first two of these would also be of use in synchronisation code for a bounded buffer used in a concurrent environment (as we show in the next example).

Because this assertion mechanism is event-based, there should not be any difficulty in combining it with a SOS-based synchronisation mechanism such as ESP. Due to the likelihood of an overlap between synchronisation variables/code and assertion variables/code, we merge the two together in order to avoid duplication. This is perfectly safe since the event-based nature of the SOS paradigm ensures that there is no possibility of, say, synchronisation code accessing a variable while it is being concurrently updated by assertion code.

A Concurrent Bounded Buffer with Assertions

The code in Figure 14.6 illustrates a bounded buffer that contains both assertions and synchronisation code. Two small syntactic points to note are as follows. Firstly, the keyword **synchronisation** has been replaced with **constraints** since the code contained in the second part of the class is for both sequential constraints (assertions) and synchronisation constraints. Secondly, the syntax used to denote guards has been changed so as to make it more consistent with that used to denote assertions.

Some more interesting points to note are as follows.

Firstly, synchronisation guards and sequential assertions happily co-exist in an object that permits internal concurrency. This is important since, as said earlier, Eiffel-style pre- and postconditions are incompatible with internal concurrency.

Secondly, the *post_term_assertions* are more complex than their counterparts in the sequential version of the bounded buffer (Figure 14.5). This is because the buffer cannot

```
class Buffer[elem] {
  ...   // instance variables
  Buffer(int Size) { ... }   // constructor
  Put(...) { ... }
  Get(...) { ... }
constraints
  int   Size, num;
  int   old_num local to Put, Get;
  int   old_term_get local to Put;
  int   old_term_put local to Get;
  start(Buffer) → { Size := this_inv.Size; num := 0; }

  guard(Put): exec(Put) = 0 and num < Size;
  pre_start_assertion(Put): num < Size;
  start(Put) → { this_inv.old_num := num; this_inv.old_term_get := term(Get); }
  term(Put) → { num ++; }
  post_term_assertion(Put): num = this_inv.old_num + 1
                                  − (term(Get) − this_inv.old_term_get);
  guard(Get): exec(Get) = 0 and num > 0;
  pre_start_assertion(Get): num > 0;
  start(Get) → { this_inv.old_num := num; this_inv.old_term_put := term(Put); }
  term(Get) → { num −−; }
  post_term_assertion(Get): num = this_inv.old_num − 1
                                  + (term(Put) − this_inv.old_term_put);
}
```

Figure 14.6: Event-based constraints for a concurrent Bounded Buffer

guarantee that, say, there will be one more item in the buffer at the termination of a *Put* invocation than there was before it started—it is possible that while a *Put* invocation is executing, one or more *Get* invocations will execute, thus reducing the number of items in the buffer. As such, the termination assertion is weaker (and more complex). The code maintains a variable, *old_term_get*, local to each *Put* invocation. This records the value of the *term*(Get) counter when the *Put* invocation starts to execute and is used to calculate how many *Get* invocations terminated while the *Put* invocation was executing. This value is then used in the *term* assertion for *Put*. The *term* assertion for *Get* is similarly complex.

Note that this increased complexity in the *term* constraints is not intrinsic to the constraint mechanism itself but rather reflects the fact that it is more difficult to reason about concurrent programs than sequential ones.

A final point to note about this buffer is the similarity between the guards and the

pre_start_assertions for the two operations. These assertions are redundant since their truth is guaranteed by successful evaluations of the guards. However, it is possible that such redundancy might be useful. For example, the bounded buffer in Figure 14.6 is suitable for use in a concurrent environment. If the guards were omitted then it would still retain its assertions and be suitable for use in a sequential environment. If a language had the capability to compile two versions of a class—one with guards enabled and one with them disabled but sequential assertions still enabled—then this would permit programmers to write classes that are reusable in both sequential and concurrent environments.[3]

The Dining Philosophers with Assertions

The examples so far have illustrated the use of assertions at the *start* and *term* events. In this next example, we illustrate how assertions at *arrival* events can also be useful.

```
class Table {
  Table(int Size) { ... }
  Eat(int pos) { ... }
constraints
  int  Size;
  start(Table) → { Size := this_inv.Size; }
  boolean ShareForks(int i, int j) { ... }
  pre_arrival_assertion(Eat): 0 <= this_inv.pos and this_inv.pos < Size;
  guard(Eat): there_is_no(p executing Eat: ShareForks(p.pos, this_inv.pos));
}
```

Figure 14.7: Dining Philosophers with assertions to check parameter range

The code in Figure 14.7 implements the Dining Philosophers exercise. The constructor takes a parameter, *Size*, indicating how many seats are at the table. The constraints code records its value and uses it in an *arrival* assertion to immediately reject any invocation of *Eat* that contains an out-of-range *pos* parameter.

14.5.5 Related Work

We said in Section 14.5.3 that although it is widely know that Eiffel-style assertions and guards are similar, there has been little written about this in the literature. However, a few papers do deal with this topic. We give a brief overview of them here as an aid to readers interested in this area.

[3] Actually, a class that contains guards *is* usable in a sequential environment. However, its behaviour in the face of an errant client would be poor. For example, if a client tried to invoke *Get* on an empty buffer then it would "hang" indefinitely rather than raise an exception to signal the error.

Proposed extension to Eiffel. Meyer [Mey93] proposes a concurrency extension for the Eiffel language. Recognising the similarity between preconditions and guards, Meyer overloads the semantics of preconditions so that they can have their traditional abort semantics when used in sequential objects, or can have delay semantics (in effect, be guards) when used in concurrent objects. Concurrency in Meyer's proposal comes by the ability of a program to be split over several address spaces ("processors" in the parlance of Meyer's paper), each with its own thread of control. Thus there can be concurrency *between* address spaces but there can not be any concurrency *within* an object.

The lack of concurrency within objects means that there is no need to specify, say, mutual exclusion constraints and thus, as discussed in Section 14.5.3, guards will often look identical to preconditions. Under these circumstances it would seem that the choice to endow preconditions with dual semantics (abort and delay) is a good one. However, Meyer's proposal is not without drawbacks. In particular, no attempt has been made to increase the expressive power of the preconditions—they can access only instance variables and parameters of the operation being invoked. Thus, as a synchronisation mechanism, this proposal has poor expressive power. For instance, the inability to compare parameters, or the relative arrival times, of other pending invocations means that there are many scheduling policies that it cannot implement easily, if at all.

CEiffel. In the CEiffel language [Löh93], a compiler option can be toggled which determines whether the synchronisation code of the class being compiled will be processed or silently ignored. The purpose of this is to be able to obtain both concurrent and sequential versions of a class from the same source file in order to promote code re-usability.

One of the synchronisation constructs in CEiffel splits an Eiffel-style precondition in two. If the compiler is directed to ignore synchronisation code then this synchronisation construct is ignored and the precondition is compiled as a traditional Eiffel-style precondition. However, if the compiler is directed to process synchronisation constructs then this construct tells the compiler to treat the first part of the precondition as being akin to (what we have termed) a *pre_arrival_assertion* and the second part as being a guard. The *pre_arrival_assertion* part is typically used to check the validity of parameters.

The CEiffel language permits concurrency within objects but does not take any precautions to ensure that instance variables are in a consistent state when assertions are being evaluated.

14.6 Summary

This chapter has discussed several possible areas for future work. Most of these are ways in which the Sos paradigm might be extended to integrate synchronisation with other language concepts. We feel that this potential for extension demonstrates the flexibility and power of the Sos paradigm.

Chapter 15

Conclusions

The main text of this thesis is split into three parts: Part II defines the Sos paradigm; Part III shows that language support for generic synchronisation policies is feasible; and Part IV analyses the ISVIS conflict which can hinder the reuse of code in COOPLs. The main contributions of each of these three parts (summarised in Sections 15.1 through to 15.3) stand by themselves. However, that is not to say that these three parts are unrelated.

Part II and Part III are related because some of the benefits of the Sos paradigm make it feasible to provide language support for GSPs without the severe limitations that has characterised previous attempts to provide such support.

Our survey of inheritance techniques to reduce the harmful effects of the ISVIS conflict shows that separating synchronisation code from sequential code is important. Since the Sos paradigm offers this separation of code, we see that Parts II and IV are related.

Finally, Parts III and IV are related because the use of generic synchronisation policies considerably reduces the harmful effects of the ISVIS conflict.

Sections 15.1 through to 15.3 summarise the main contributions of each of Parts II, III and IV, respectively.

15.1 The Sos paradigm

The central contribution of the Sos paradigm is the excellent expressive power that it offers. More particularly, it offers this power without any of the undesirable traits often associated with expressive power, as we now discuss.

Sos is based on just four concepts. As such, Sos-based mechanisms can avoid complexity. This is in stark contrast to many other mechanisms which introduce a mass of constructs (and complexity) in an attempt to increase expressive power. The simplicity of the Sos paradigm also shows up in the ease with which we were able to introduce a Sos-based mechanism, Esp, to an existing, object-oriented language.

The Sos paradigm shows that it is possible for synchronisation code to be completely separated from sequential code. This promotes modularity and is of vital importance in

being able to provide language support for generic synchronisation policies, which in turn reduces the harmful effects of the ISVIS conflict.

In an attempt to obtain sufficient expressive power to implement some synchronisation policies, many languages permit synchronisation code to access the instance variables of an object. In languages that support concurrency within objects, this access can be dangerous since synchronisation code may examine an instance variable while it is in an inconsistent state. The Sos paradigm shows that such access is unnecessary since synchronisation code can maintain its own variables.

Finally, the Esp synchronisation mechanism, which embodies the concepts of the Sos paradigm, offers another benefit: it is simultaneously a declarative and a procedural mechanism. As such, it offers excellent expressive power and degrades gracefully.

15.2 Language Support for Generic Synchronisation Policies

There have been a few attempts in the past to provide language support for generic synchronisation policies. However, as we have shown in Section 10.1, these have entailed serious limitations: they failed to completely separate synchronisation code from sequential code and hence offered limited expressive power and/or were not fully generic; in some cases policies could only be instantiated upon individual operations rather than sets of operations; one language, Dragoon, confused genericity with inheritance and this resulted in idiosyncrasies in the type system.

With the aid of the Sos paradigm, we have overcome all of these limitations and have provided comprehensive support for generic synchronisation policies. Aside from code reuse, GSPs offer several additional benefits. Firstly, we have shown that GSPs facilitate the optimisation of synchronisation code (via the *optimisation by transformation* technique). Secondly, our analysis of the ISVIS conflict shows that its harmful effects are reduced considerably by the use of GSPs. Thirdly, the instantiation of a generic synchronisation policy is extremely declarative—even if a procedural synchronisation mechanism is used to implement the policy. As such, it seems possible that GSPs may do more to reduce the complexity of concurrent programming than expressively powerful synchronisation mechanisms. Finally, an interesting possibility is that if a COOPL provides support to *instantiate* generic synchronisation policies then it need not provide a synchronisation mechanism to actually *write* them. This, in turn, might reduce the complexity of COOPLs.

15.3 Analysis of the ISVIS Conflict

Most of the research to date on the ISVIS conflict (or the "inheritance anomaly" as Matsuoka calls it) has assumed that the problem is due to a conflict between synchronisation and inheritance. We have shown that this assumption is incorrect and that the problem is due to a conflict between two different uses of inheritance. We have conducted a survey to

determine: (i) how the ISVIS conflict manifests itself in a number of inheritance mechanisms; and (ii) what techniques are effective in reducing the harmful effects of the ISVIS conflict. We have shown that the use of generic synchronisation policies considerably reduces these harmful effects.

Bibliography

[ABvdSB94] Mehmet Aksit, Jan Bosch, William van der Sterren, and Lodewijk Bergmans. Real-time Specification Inheritance Anomalies and Real-time Filters. In Mario Tokoro and Remo Pareschi, editors, *ECOOP'94*, pages 386–405. Springer-Verlag, July 1994. Published as volume 821 of the *Lecture Notes in Computer Science* series.

[ACR92] Colin Atkinson and Stefano Crespi-Reghizzi. Behavioural Inheritance: Themes and Variations. Presented at the ECOOP'92 workshop on *Object-based Concurrency and Reuse*, 1992.

[Ada83] The Programming Language Ada Reference Manual. Published in: Lecture Notes in Computer Science, Vol 155, Springer-Verlag, 1983.

[AGMB91] Colin Atkinson, Stephen Goldsack, Andrea Di Maio, and Rami Bayan. Object oriented Concurrency and Distribution in DRAGOON. *Journal of Object-Oriented Programming (JOOP)*, 4(1):11–19, March/April 1991.

[Ame87] Pierre America. Inheritance and Subtyping in a Parallel Object-Oriented Language. In G. Goos and J. Hartmanis, editors, *ECOOP '87—European Conference on Object-Oriented Programming*, pages 234–242, June 1987. Published as volume 276 of the *Lecture Notes in Computer Science* series.

[And79] Sten Andler. Predicate Path Expressions. In *Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 226–236, San Antonio, Texas, 1979.

[And81] Gregory R. Andrews. Synchronising Resources. *ACM Transactions on Programming Languages and Systems*, 3(4):405–430, October 1981.

[And91] Gregory R. Andrews. *Concurrent Programming: Principles and Practice.* The Benjamin/Cummings Publishing Company, Inc., 1991.

[Atk90] Colin Atkinson. *An Object-Oriented Language for Software Reuse and Distribution.* PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, University of London, London SW7 2BZ, England, February 1990.

[Atk91] Colin Atkinson. *Object-Oriented Reuse, Concurrency and Distribution.* ACM Press and Addison-Wesley, 1st edition, 1991.

[AW93] Colin Atkinson and David Weller. Integrating Inheritance and Synchronisation in Ada9X. In *Proceedings of TriAda'93*, pages 229–241, 1993.

[BAWY92] Lodewijk Bergmans, Mehmet Aksit, Ken Wakita, and Akinori Yonezawa. An Object-oriented Model for Extensible Concurrent Systems: The Composition-Filters Approach. Submitted for publication, September 1992.

[BBI+] M. Banâtre, Y. Belhamissi, V. Issarny, I. Puaut, and J.P. Routeau. Arche: A Framework for Parallel Object-oriented Programming Above a Distributed Architecture. To appear in the Proceedings of the 14th International Conference on Distributed Computing Systems, 1994.

[BC90] Gilad Bracha and William Cook. Mixin-based Inheritance. In Norman Meyrowitz, editor, *ECOOP/OOPSLA'90 Proceedings*, pages 303–311. ACM Press, 21–25 October 1990. Special Issue of *SIGPLAN Notices*, 25(10).

[Ber94] Lodewijk Bergmans. *Composing Concurrent Objects: Applying Composition-filters for the Development and Reuse of Concurrent Object-oriented Programs.* PhD thesis, TRESE (Twente Research and Education on Software Engineering), The SETI Group, Department of Computer Science, University of Twente, Enschede, The Netherlands, June 1994.

[BFS93] J.P. Bahroun, L. Féraud, and J.C. Sakdavong. Designing and Implementing Synchronisation: An Object-oriented Approach. In *TOOLS Europe'93*, pages 235–248. Prentice Hall, 1993.

[BH78] Per Brinch Hansen. Distributed Processes: A Concurrent Programming Concept. *Communications of the ACM*, 21(11):934–941, November 1978.

[BI92] Marc Benveniste and Valérie Issarny. Concurrent Programming Notations in the Object-oriented Language Arche. Rapport de Recherche 1822, IRISA (Institut de Recherche en Informatique et Systemes Aleatoires), Campus Univeritaire de Beaulieu, 35042 - Rennes Cédex, France, December 1992.

[Blo79] Toby Bloom. Evaluating Synchronisation Mechanisms. In *Seventh International ACM Symposium on Operating System Principles*, pages 24–32, 1979.

[BY87] Jean-Pierre Briot and Akinori Yonezawa. Inheritance and Synchronisation in Concurrent OOP. In *ECOOP'87*, pages 35–43, 15–17 June 1987. Published as Issue 54 of *BIGRE*.

[Cam83] R. H. Campbell. Distributed Path Pascal. In Y. Parker and J. P. Verjus, editors, *Distributed Computer Systems: Synchronisation, Control and Communication*, pages 191–223. Academic Press, 1983.

[Car90a] Denis Caromel. Concurrency: An Object-Oriented Approach. In Jean Bézivin, Bertrand Meyer, and Jean-Marc Nerson, editors, *TOOLS 2 (Technology of Object-Oriented Languages and Systems)*, pages 183–197. Angkor, 1990.

[Car90b] Denis Caromel. Programming Abstractions for Concurrent Programming. In *Technology of Object-Oriented Languages and Systems, PACIFIC (TOOLS PACIFIC '90)*, November 1990. Also internal report 90-R-107, Centre de Recherche en Informatique de Nancy, Vandoeuvre-Lès-Nancy, 1990.

[Car93] Denis Caromel. Toward a Method of Object-oriented Concurrent Programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[CGM91] Luc Courtrai, Jean-Marc Geib, and Jean-François Méhaut. Inheritance of Synchronisation Constraints in the VCP System. In Augustin Mrazik, editor, *Proceedings of EastEurOOpe'91*, pages 57–60, September 1991.

[CH73] R. H. Campbell and A. N. Habermann. The Specification Of Process Synchronisation by Path Expressions. In *Lecture Notes in Computer Science, No. 16*, pages 89–102. Springer Verlag, 1973.

[CK80] Roy H. Campbell and Robert B. Kolstad. An Overview of Path Pascal's Design and the Path Pascal User Manual. *ACM SIGPLAN Notices*, 15(9):13–24, September 1980.

[CL91] A. Corradi and L. Leonardi. PO Constraints as Tools to Synchronise Active Objects. *Journal of Object-oriented Programming (JOOP)*, 4(6):41–53, October 1991.

[Cou94] Gary Couse. Considerations in Reducing Interference in Concurrent Object-oriented Programming Languages. Master's thesis, Department of Computer Science, University College, Cork, Ireland, June 1994.

[DDR+91] D. Decouchant, P. Le Dot, M. Riveill, C. Roisin, and X. Rousset de Pina. A Synchronisation Mechanism for an Object Oriented Distributed System. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 152–159, Arlington, Texas, 20–24 May 1991. IEEE.

[Dij75] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[DKM+88] D. Decouchant, S. Krakowiak, M. Meysembourg, M. Riveill, and X. Rousset de Pina. A Synchronisation Mechanism for Typed Objects in a Distributed System. Presented at the workshop on Object-Based Concurrent Programming, OOPSLA '88, San Diego, September 1988. Abstract in *ACM SIGPLAN Notices*, 24(4), pages 105–107, April 1989.

[Frø92] Svend Frølund. Inheritance of Synchronisation Constraints in Concurrent Object-oriented Programming Languages. In Ole Lehrmann Madsen, editor, *Proceedings of ECOOP'92*, pages 185–196. Springer-Verlag, 29 June to 3 July 1992. Published as volume 615 of the *Lecture Notes in Computer Science* series.

[GC86] J. E. Grass and R. H. Campbell. Mediators: A Synchronisation Mechanism. In *Proceedings of the Conference on Distributed Computer Systems*, pages 468–477. IEEE, September 1986.

[GC92] Jean-Marc Geib and Luc Courtrai. Abstractions for Synchronisation to Inherit Synchronisation Constraints. Presented at the ECOOP'92 workshop on Object-based Concurrency and Reuse, June 1992.

[Ger77] A. J. Gerber. Process Synchronisation by Counter Variables. *ACM Operating Systems Review*, 11(4):6–17, October 1977.

[GMC+89] Stefano Genolini, Andrea Di Maio, Cinzia Cardigno, Stephen Goldsack, and Colin Atkinson. Specifying Synchronisation Constraints in a Concurrent

Object-Oriented Language. In *First International Conference on Technology of Object-Oriented Languages and Systems (TOOLS)*, 1989.

[Gro90] Peter Grogono. The Book of Dee. Technical Report OOP-90-3, Department of Computer Science, Concordia University, 1455 deMaisonneuve Blvd. West, Montréal, Québec, Canada H3G 1M8, 1990.

[Gro92] The Distributed Systems Group. The C** Programmer's Guide. Technical Report TCD-CS-92-03, Department of Computer Science, Trinity College, Dublin 2, Ireland, February 1992.

[GW91] Wolfgang Gerteis and Wolfgang Wiraz. Synchronising Objects by Conditional Path Expressions. In *TOOLS Pacific'91*, pages 193–201, 1991.

[Hal85] R. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[Hoa74] C.A.R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[KB93] Murat Karaorman and John Bruno. Introducing Concurrency to a Sequential Language. *Communications of the ACM*, 36(9):103–116, September 1993.

[KL89] Dennis G. Kafura and Keung Hae Lee. Inheritance in Actor Based Concurrent Object-Oriented Languages. In Stephen Cook, editor, *ECOOP 89*, pages 131–145. Cambridge University Press, July 1989.

[KMMPN87] B. B. Kristensen, O. L. Madsen, B. Moller-Pedersen, and K. Nygaard. *Research Directions in Object-oriented Programming*, chapter The Beta Programming Language, pages 7–48. MIT Press, 1987.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[LL94] Cristina Videira Lopes and Karl J. Lieberherr. Abstracting Process-to-Function Relations in Concurrent Object-Oriented Applications. In Mario Tokoro and Remo Pareschi, editors, *ECOOP'94*, pages 81–99. Springer-Verlag, July 1994. Published as volume 821 of the *Lecture Notes in Computer Science* series.

[Löh91] Klaus-Peter Löhr. Concurrency Annotations and Reusability. Report B-91-13, Frachbereich Mathematik, Freie Universität, Berlin, November 1991.

[Löh93] Klaus-Peter Löhr. Concurrency Annotations for Reusable Software. *Communications of the ACM*, 36(9):81–89, September 1993.

[LR80] Butler W. Lampson and David D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[LSAS77] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction Mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[Mat93] Satoshi Matsuoka. *Language Features for Re-use and Extensibility in Concurrent Object-oriented Programming Languages.* PhD thesis, Department of Information Science, The University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan, April 1993.

[MBWD94] Ciaran McHale, Seán Baker, Bridget Walsh, and Alexis Donnelly. Synchronisation Variables. Technical Report TCD-CS-94-01, Department of Computer Science, Trinity College, Dublin 2, Ireland, January 1994.

[MCB+89] Andrea Di Maio, Cinzia Cardigno, Rami Bayan, Catherine Destombes, and Colin Atkinson. DRAGOON: An Ada-based Object Oriented Language for Concurrent, Real-Time, Distributed Systems. In *Ada-Europe International Conference*, Madrid, Spain, June 1989.

[McH89] Ciaran McHale. Pasm: A Language for Teaching Concurrency. BA project report, Department of Computer Science, Trinity College, Dublin 2, Ireland, April 1989.

[Mes93] José Meseguer. Solving the Inheritance Anomaly in Concurrent Object-oriented Programming. In Oscar M. Nierstrasz, editor, *ECOOP'93*, pages 220–246. Springer-Verlag, July 1993. Published as volume 707 of the *Lecture Notes in Computer Science* series.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

[Mey92] Bertrand Meyer. *Eiffel: The Language.* Prentice Hall, 1992.

[Mey93] Bertrand Meyer. Systematic Concurrent Object-oriented Programming. *Communications of the ACM*, 36(9):56–80, September 1993.

[MK87] J. Eliot B. Moss and Walter H. Kohler. Concurrency Features for the Trellis/Owl Language. In G. Goose and J. Hartmanis, editors, *ECOOP '87—European Conference on Object-Oriented Programming*, pages 171–180. Springer-Verlag, June 1987. Published as volume 276 of the *Lecture Notes in Computer Science* series.

[MTY93] Satoshi Matsuoka, Kenjiro Taura, and Akinori Yonezawa. Highly Efficient and Encapsulated Re-use of Synchronisation Code in Concurrent Object-Oriented Languages. In *Proceedings of OOPSLA'93*, pages 109–126, October 1993. Published as a special issue of *SIGPLAN Notices*, 28(10).

[MWBD91] Ciaran McHale, Bridget Walsh, Seán Baker, and Alexis Donnelly. Scheduling Predicates. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing*, pages 177–193, July 1991. Published as volume 612 of Lecture Notes in Computer Science. Springer-Verlag. Also available as technical report TCD-CS-91-24, Department of Computer Science, Trinity College, Dublin 2, Ireland.

[MWBD92] Ciaran McHale, Bridget Walsh, Seán Baker, and Alexis Donnelly. Evaluating Synchronisation Mechanisms: The Inheritance Matrix. Technical Report TCD-CS-92-18, Department of Computer Science, Trinity College, Dublin 2, Ireland, July 1992. Presented at the ECOOP'92 Workshop on *Object-based Concurrency and Reuse*.

[MWY90] Satoshi Matsuoka, Ken Wakita, and Akinori Yonezawa. Synchronisation Constraints With Inheritance: What Is Not Possible—So What Is? Technical Report 90-010, Department of Information Science, The University of Tokyo, 7-3-1 Hongo, Bunkyo-Ku, Tokyo 113, Japan, 1990.

[MY90] Satoshi Matsuoka and Akinori Yonezawa. Metalevel Solution to Inheritance Anomaly in Concurrent Object-Oriented Languages. Presented at the OOPSLA/ECOOP'90 Workshop on Reflections and Metalevel Architectures in Object-Oriented Languages, 1990.

[MY93] Satoshi Matsuoka and Akinori Yonezawa. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, chapter 1, pages 107–150. MIT Press, 1993.

[Neu91] Christian Neusius. Synchronising Actions. In Pierre America, editor, *ECOOP '91*, pages 118–132, Geneva, Switzerland, July 1991. Springer-Verlag. Published as volume 512 of the *Lecture Notes in Computer Science* series.

[Nie87] O. M. Nierstrasz. Active Objects in Hybrid. In Norman Meyrowitz, editor, *OOPSLA '87 Proceedings*. ACM, 1987. Special issue of *ACM SIGPLAN Notices*, 22(12):243–253.

[Pap93] Michael Papathomas. *Language Design Rationale and Semantic Framework for Concurrent Object-oriented Programming*. PhD thesis, Department d'Informatique, Université de Genève, 1993.

[RdPG91] S. Crespi Reghizzi, G. Galli de Paratesi, and S. Genolini. Definition of Reusable Concurrent Software Components. In Pierre America, editor, *ECOOP '91*, pages 148–166, Geneva, Switzerland, July 1991. Springer-Verlag. Published as volume 512 of the *Lecture Notes in Computer Science* series.

[Riv92] Michel Riveill. An Overview of the Guide Language. Presented at the OOPSLA'92 workshop on Objects in Large Distributed Applications, 1992.

[RR92] Michel Reveill and Xavier Rousset. Reusable Synchronised Objects. Presented at the ECOOP'92 workshop on *Object-based Concurrency and Reuse*, 1992.

[RV77] Pierre Robert and Jean-Pierre Verjus. Towards Autonomous Descriptions of Synchronisation Modules. In Bruce Gilchrist, editor, *Information Processing 77: Proceedings of the IFIP (International Federation of Information Processing) Congress 77*, pages 981–986. North-Holland Publishing Company, 8–12 August 1977.

[Sel90] Robert Seliger. Extending C++ to Support Remote Procedure Call, Concurrency, Exception Handling, and Garbage Collection. In *USENIX C++ Conference Proceedings*, pages 241–261, San Francisco, California, 9–11 April 1990.

[Sny86] Alan Snyder. Encapsulation and Inheritance in Object-oriented Programming Languages. In Norman Meyrowitz, editor, *OOPSLA'86 Proceedings*, pages 38–45, nov 1986. Special issue of *SIGPLAN Notices*, 21(11).

[Ste87] Lynn Andrea Stein. Delegation Is Inheritance. In Norman Meyrowitz, editor, *Proceedings of OOPSLA'87*, pages 138–146, December 1987. Published as a special issue of *SIGPLAN Notices*, 22(12).

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[TA88] Anand Tripathi and Mehmet Aksit. Communication, Scheduling, and Resource Management in SINA. *JOOP (Journal of Object Oriented Programming)*, pages 24–37, November 1988.

[Tho92] Laurent Thomas. Extensibility and Reuse of Object-oriented Synchronisation Components. In *PARLE'92 (Parallel Architectures & Languages Europe)*, pages 261–275. Springer Verlag, 1992. Published as volume 605 in the Lecture Notes in Computer Science series.

[Tho94] Laurent Thomas. Inheritance Anomaly in True Concurrent Object-oriented Languages: A Proposal. In *IEEE TENCON'94*, pages 541–545, August 1994.

[TM93] Kenjiro Taura and Satoshi Matsuoka. An Efficient Implementation Scheme of Concurrent Object-Oriented Languages on Stock Multiprocessors. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 218–228, July 1993. Published as a special issue of *SIGPLAN Notices*, 28(7).

[TS89] Chris Tomlinson and Vineet Singh. Inheritance and Synchronisation with Enabled-Sets. In *OOPSLA '89 Proceedings*, pages 103–112, October 1989.

[Zen90] Steven Ericsson Zenith. Linda coordination language: subsystem kernel architecture (on transputers). Research Report YALEU/DCS/RR-794, Yale University, Department of Computer Science, New Haven, Connecticut. USA, 29 May 1990.