Scheduling Predicates

Ciaran McHale[1], Bridget Walsh,
Seán Baker, Alexis Donnelly

Abstract

In this report, we present a powerful new synchronisation mechanism called *scheduling predicates*. These predicates—*there_are_no*, *there_exists* and *for_all*—allow the programmer to schedule the order of execution of operations based on relative arrival times, values of parameters, and built-in synchronisation counters.

Since many synchronisation problems are, in fact, scheduling problems, these facilitate much simpler and clearer solutions to such problems. We also show that this mechanism subsumes and unifies the existing declarative synchronisation mechanisms used in some object-oriented languages, and extends the number of problems for which a purely declarative approach is possible.

1 Introduction

There has been extensive research into developing powerful synchronisation mechanisms. The approaches taken generally fall into one of two categories.

Firstly are procedural mechanisms which combine synchronisation primitives with sequential flow control constructs and data structures, thus allowing the programmer to implement synchronisation policies by algorithmic means.

In contrast to these are declarative mechanisms in which the programmer simply specifies the synchronisation policy desired. An ideal declarative mechanism would have the desirable property that the specification of a synchronisation policy *is* its implementation. When the power of a declarative mechanism is sufficient for a particular policy, the solution is usually trivial.

However, declarative mechanisms are currently limited in their power in that they can only directly specify simple policies.

For more complex policies, the programmer must fall back to a procedural approach which, when implemented in the declarative notation, results in quite inelegant solutions. The use of *synchronisation procedures*[2] in Path Expressions [8] is a prime example of this; the Guide solution to the FIFO variant of the readers/writer problem [11] is another example of the need for synchronisation procedures when a declarative mechanism has insufficient power to express a synchronisation policy directly.

[1]Authors' address: Department of Computer Science, Trinity College, Dublin 2, Ireland.
Tel: +353-1-7021539 Email: {cjmchale,bwalsh,baker,donnelly}@cs.tcd.ie

[2]This term was coined by Bloom [6].

Our approach is to increase the power of declarative mechanisms so that they can easily solve a wider range of synchronisation policies: in particular, policies which *schedule* as well as *synchronise* requests. This report takes a step in this direction by presenting the concept of *scheduling predicates.* Rather than being just "yet another synchronisation mechanism" with its own strengths and weaknesses, *scheduling predicates* will be shown to unify several other synchronisation mechanisms. As such it is both an improvement on, and a generalisation of existing mechanisms.

1.1 Syntax Used in Report

The syntax used in this report (see Figure 1) should, for the most part, be self-explanatory. Comments are denoted by "//" and last until the end of the line. Macros are declared by use of the "**#define**" C preprocessor directive.

class <class_name> **is**
    <declaration of instance variables>
    <declaration of operations Op_1, Op_2, ...>
    entry guards
    Op_1: <a guard>;
    Op_2: <another guard>;
        ⋮
end <class_name>;

Figure 1: Syntax used in this report

The synchronisation constraints are expressed in the **entry guards** clause which appears after all of the class' operations. An **entry guard** is a boolean expression which must become true before a request to execute an operation on an object is permitted to continue. Several languages employ the concept of guards, in one form or another, in their synchronisation mechanisms: CSP [15], Synchronising Resources (SR) [4], Ada [1], Guide [10], Mediators [13] DRAGOON [12] and Predicate Path Expressions [2] are the best known examples. Scheduling predicates improve upon these previous mechanisms by having more expressively powerful guards.

1.2 Structure of Report

This report is structured as follows:

Section 2 presents a brief overview of the different object models which can be commonly found in concurrent object-oriented languages. Section 3 then discusses some existing declarative synchronisation mechanisms and their limitations. Section 4 introduces *scheduling predicates*, and an analysis of their power is presented in section 5. This analysis shows up a weakness of the mechanism which is discussed in section 6. Section 7 then shows how scheduling predicates subsume the mechanisms discussed in section 3. Section 8 discusses implementation and optimisation issues. Section 9 discusses the problem of synchronisation counter overflow. Finally, section 10 concludes the report and mentions some areas in which we intend to work in future.

2 Object Models

In some concurrent object-oriented languages (e.g., Guide [11]), objects are *passive* entities which reside in shared memory and may be accessed simultaneously by several processes. This model can support concurrency within an object.

In other systems, objects are deemed to be *active*: they have a thread of control which receives requests and schedules them for service. Some active object models (e.g., Hybrid [18] and Caromel [9]) have just a single thread within an object which both schedules and services requests. This object model does not support concurrency *within* objects, though usually some form of inter-object concurrency is supported.

Other active object models do support internal concurrency. For example, an object may have a thread of control which receives and schedules requests; and when a request may start execution, another thread is created to service it [17].

Most mechanisms have inherent limitations which makes then suitable for only one object model, and less suitable, or even totally unsuitable, for others. For example, Caromel's mechanism [9] provides good support for scheduling requests; however it provides no facilities to manage internal concurrency, thus limiting its usability to single-threaded, active objects. Conversely, the main strength of synchronisation counters [11, 12] is in managing internal concurrency rather than scheduling requests; so while they are suitable for passive object models, they would be of limited use in the single-threaded, active object model.

The mechanism presented in this paper does not have this limitation: since it provides good facilities to both manage internal concurrency and schedule pending requests, it is effective in both passive and active object models.[3] Being able to deal with synchronisation and scheduling in a uniform manner across different object models reduces the learning curve for programmers who must move between different environments, and it also promotes reuse of concurrent code.

3 Limitations of Existing Declarative Mechanisms

3.1 Synchronisation Counters

Synchronisation counters [3] feature heavily in declarative mechanisms such as the Guide and DRAGOON languages. These, automatically maintained, variables of an object count the total number of invocations for each operation of the object that have been *requested*, have *started* execution and have *terminated* execution etc. There are actually five counters for each operation on an object. These are:[4]

$req(Op)$	: number of requests to execute operation Op.
$wait(Op)$	: number of requests that are waiting to execute Op.
$start(Op)$	: number of requests that have started executing Op.
$exec(Op)$	: number of requests that are currently executing Op.

[3]The aspects of our mechanism which deal with internal concurrency of an object are, by definition, redundant in the single-threaded, active object model.

[4]These are the counter names used throughout this report. Unfortunately, there are no standard names for these counters; for example, both Guide and DRAGOON use different names.

$term(Op)$: number of requests that have terminated execution of Op.

Only three of these need to be stored since the following relationships hold:

$$wait(Op) = req(Op) - start(Op)$$
$$exec(Op) = start(Op) - term(Op)$$

Synchronisation counters may be used in guards. Although they have been used most extensively in languages which embody the passive object model, they could also be used in active object models. However, in the single-threaded, active object model the *exec* counter becomes redundant.

All of the synchronisation counters are initialised to zero and the compiler produces the necessary code to ensure that they are updated atomically. An example of their usage is given in Figure 2 which solves the bounded buffer problem. The reader is referred elsewhere [5, 10] for further examples illustrating the use of synchronisation counters.

```
entry guards
#define not_full = term(Put) - term(Get) < size
#define not_empty = term(Put) - term(Get) > 0
Put: exec(Put) = 0 and not_full;
Get: exec(Get) = 0 and not_empty;
```

Figure 2: Solution to the Bounded Buffer problem

Synchronisation counters have their limitations. In particular, they provide no way to *schedule* requests based on either request parameters or arrival time. Thus, though Guide and DRAGOON can easily solve the simpler variations of the readers/writer problem (such as readers' priority or writer's priority), synchronisation counters are of little assistance in solving the FIFO variant. The complexity of the Guide solution to this problem [11] clearly shows this limitation. Other scheduling problems, such as Shortest Job Next [7] and the Disk Head Scheduler [14] are probably impossible to solve with synchronisation counters alone.

3.2 Scheduling with the "By" Clause

A step towards providing scheduling support is the **by** clause as provided in Synchronising Resources (SR) [4]: this can be used to schedule requests based on one of their parameters. The **by** keyword is followed by an arithmetic expression whose value determines the order in which invocations are to be executed (minimum value first). Figure 3 shows (in the notation of this report) the **by** clause implementation of the Shortest Job Next Scheduler. The **by** clause is noteworthy in that it provides a declarative way to schedule requests based on their parameters. However, it has the following limitations:

1. Requests cannot be scheduled based on their relative arrival times.
2. Scheduling may only be easily performed based on a single parameter.[5]

[5]If scheduling based on two parameters is required then a **by** clause of the form

3. It may only be used to schedule requests for the same operation. Thus, for example, it could not be used to implement the FIFO variant of the readers/writer problem in which the scheduling of a *Write* request is dependent not only on other *Write* requests, but also on *Read* requests; and where scheduling of a *Read* request is dependent on *Write* requests.

The first limitation could be easily overcome by having the compiler implicitly associate an *arrival* time attribute with each request. If this were done then a FIFO scheduler could be implemented with "**by** *arrival*". The second limitation would be overcome if a list of expressions could follow the **by** keyword. For example, "**by** *len*, *arrival*" would implement a SJN scheduler with FIFO sub-ordering, i.e., if several requests have equal *len* parameters then they will be served in *arrival* order. However it is hard to imagine any variation of the **by** clause which could overcome the third limitation.

```
class Printer is
    Print(len: Integer, FileName: String ) is ...

    entry guards
    Print: exec(Print) = 0 by len;
End Printer;
```

Figure 3: Shortest Job Next

4 Scheduling Predicates

Consider the basic readers/writer problem. An English description of this synchronisation policy might be:

> *Read* may execute if there are currently no executions of *Write*.
>
> *Write* may execute if there are currently no executions of either *Read* or *Write*.

A solution to this using synchronisation counters is:

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

Note that this solution closely mirrors the English description of the problem. This is an excellent example of the desirable "specification is the implementation" property of declarative mechanisms. Unfortunately, synchronisation counters provide little support to *schedule*, as opposed to just *synchronise*, requests. Thus, for many scheduling

---

by $\text{param}_1 * \text{scale}_1 + \text{param}_2$

where scale_1 is one higher than the maximum value of param_2. (This can obviously be extended to allow scheduling on any number of parameters.) Problems with this approach are (i) the meaning is not immediately obvious; and (ii) there must be a fixed upper bound on param_2.

problems, a solution using synchronisation counters bears little resemblance to the specification of the problem. As an example, consider the following English specification of the "Shortest Job Next" scheduler:

> *Print* may execute if (i) there are no current executions of *Print* and (ii) there are no requests waiting to execute *Print* which have a smaller *len* parameter.

Synchronisation counters can be used to implement part (i) of this policy but a more powerful mechanism is required to implement part (ii) directly. We now introduce the concept of *scheduling predicates* in which part (ii) is expressed as:

> *there_are_no*(p *waiting* Print: p.len < *my_req*.len)

In reading this notation, ":" denotes "such that". The variable p is used to iterate through all the requests waiting to execute *Print*, and *my_req* denotes the request for which the guard is being evaluated. Thus we can read the above as:

> There are no requests p waiting to execute *Print* such that the *len* parameter of p is less than my own *len* parameter.

Anding this scheduling predicate with "*exec*(Print) = 0" yields a complete entry guard for the SJN scheduler.

It is frequently desired to schedule access to an object based on the arrival time of requests rather than (or sometimes, as well as) request parameters. For example, one might wish to have a FIFO print queue. Scheduling predicates handle this by implicitly associating an *arrival* time attribute with each request. *Arrival* can be likened to a ticket machine in a tax office which gives a numbered ticket to each customer that arrives [19].

```
class Printer is
      Print(len: Integer; FileName: String ) is ...

      entry guards
      Print: exec(Print) = 0 and there_are_no(p waiting Print:
                                    p.len < my_req.len or
                                    p.len = my_req.len and p.arrival < my_req.arrival);
end Printer;
```

Figure 4: Shortest Job Next with FIFO sub-ordering

Each customer has a different numbered ticket and this can be used to ensure that customers are served in order of their arrival.[6] To implement a FIFO printer queue, all that need be done is to replace *len* with *arrival* in the solution to the SJN scheduler discussed previously. Further changing of "<" to ">" would turn the FIFO scheduler

[6]The reader may be wondering if there is any relationship between *arrival* and the synchronisation counters. An implementation might maintain the relationship $arrival = \sum_{i=1}^{n} req(Op_i)$ where n is the number of synchronised operations of the object. However, an implementation is not *required* to maintain this relationship. For example, an implementation might use a high speed clock to note *arrival* times.

into a LIFO scheduler. The entry guard in Figure 4 specifies the SJN scheduler with FIFO sub-ordering.

So far, we have demonstrated two ways in which *there_are_no* is more powerful than SR's **by** clause: not only can requests be scheduled on several parameters, the relative *arrival* time of requests can also be taken into account.

Figure 5 (FIFO readers/writer) shows how interdependent scheduling of several operations may be handled just as easily. In this scheme a *Read* request may only be executed if there are no *Write* requests ahead of it. Similarly, a *Write* request may only start if there are no *Read* or *Write* requests ahead of it.

```
class ReadersWriter is
    Read: Element is ...
    Write(elem: Element) is ...

    entry guards
    Read: exec(Write) = 0 and there_are_no(w waiting Write:
                                        w.arrival< my_req.arrival);
    Write: exec(Read,Write)[7]= 0 and there_are_no(rw waiting Read,Write:
                                        rw.arrival< my_req.arrival);
end ReadersWriter;
```

Figure 5: Solution to the FIFO Readers/Writer problem

4.1 Other Scheduling Predicates

There_are_no is actually syntactic sugar for the more fundamental function *count*. *Count* returns an integer indicating how many outstanding requests satisfy *count*'s boolean condition. The following equality holds:

$$\begin{aligned}&\textit{there\_are\_no}(\text{var\_id } \textit{waiting operation\_id} : \textit{boolean expr})\\&\quad\equiv \textit{count}(\text{var\_id } \textit{waiting operation\_id} : \textit{boolean expr}) = 0\end{aligned}$$

The companion predicates *there_exists*, and *for_all* are also available to the programmer for which the following equalities hold:

$$\begin{aligned}&\textit{there\_exists}(\text{var\_id } \textit{waiting operation\_id} : \textit{boolean expr})\\&\quad\equiv \textit{count}(\text{var\_id } \textit{waiting operation\_id} : \textit{boolean expr}) > 0\end{aligned}$$

$$\begin{aligned}&\textit{for\_all}(\text{var\_id } \textit{waiting operation\_id} : \textit{boolean expr})\\&\quad\equiv \textit{count}(\text{var\_id } \textit{waiting operation\_id} : \textit{boolean expr}) = \textit{wait}(\textit{operation\_id})\end{aligned}$$

As an example of the use of *for_all*, consider the following entry guard which implements the Shortest Job Next scheduling policy:

Print: $\textit{exec}(\text{Print}) = 0$ **and** $\textit{for\_all}(\text{p } \textit{waiting} \text{ Print: p.arrival} \geq \textit{my\_req}.\text{arrival})$;

It is trivial to express the same scheduling policy using *there_are_no*: simply replace "$\geq$" with "$<$". This interchangeable nature of the scheduling predicates gives the programmer the freedom to express a desired scheduling policy with whichever predicate feels the most natural for the task.

[7]Note that the form $exec(A, B, \ldots, Z)$ is a shorthand for $exec(A) + exec(B) + \cdots + exec(Z)$.

4.2 Other Scheduling Verbs

All the examples of scheduling predicates so far have made use of the *waiting* "verb", which is related to the *wait* counter. The other counters also have related "verbs":[8] *requested*, *started*, *executing*[9] and *terminated*.

The *executing* verb can be shown to good effect in solving the Dining Philosophers 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 *Eat*ing to her left, to her right at or at the table position she wants to use. This is implemented directly, using the *executing* verb, in Figure 6. Note, however, that this solution does not guarantee against starvation of a philosopher by conspiracy on the part of the others to keep her blocked. The random eating and thinking times of philosphers would presumably rule this out, but if such a guarantee is required then this can be easily derived by employing the *waiting* verb and *arrival* in addition to *executing*.

```
class Table is
    Eat(pos: 0..4) is ...

    entry guards
    #define RightOf(k)        ((k + 1) mod 5)
    #define ShareForks(i, j)        ( RightOf(i) = j or i = j or RightOf(j) = i )
    Eat: there_are_no(p executing Eat: ShareForks(p.pos, my_req.pos));
end Table;
```

Figure 6: Solution to Dining Philosophers problem using the *executing* verb

4.2.1 Modification of Parameters in Operations

The introduction of scheduling verbs other than *waiting* raises an issue which needs to be resolved.

Many languages (e.g., C and Pascal) allow call-by-value parameters to be modified in the body of operations.[10] Consider the code in Figure 7; the entry guard references the i parameter of requests currently executing *Bar*. However, this parameter is modified in the body of *Bar* which raises the question of which value of i the entry guard should see: the *original* value of i (i.e., the value at the time the request was received), or its *current* value?

The semantics we have chosen is that entry guards always see the *original* value of parameters. This has several advantages over entry guards seeing the most up-to-date value:

- If entry guards saw the most up-to-date value of parameters then this would reduce the separation between the implementation of operations and the synchronisation over them, thus breaking one of Bloom's [6] modularity requirements.

[8]This relationship between synchronisation counters and scheduling predicate "verbs" is defined in Section 7.1.

[9]Like the *exec* counter, the *executing* verb is redundant in the single-threaded, active object model.

[10]Some other languages, such as Ada, do not allow this.

- Efficient implementation is likely to be easier since entry guards do not need to be continually re-evaluated if the body of an operation updates a parameter referenced in an entry guard.

```
class Foo is
    Bar(i: integer) is
    begin
        ...
        i := ...;
        ...
    end Bar;

    entry guards
    Bar: there_are_no(p executing Bar: p.i = my_req.i);
end Foo;
```

Figure 7: Modification of parameters which appear in entry guards

4.2.2 Implementation Concerns of the Scheduling Verbs

It should be noted that, in general, it would be infeasible for an implementation to support the *requested*, *started* and *terminated* verbs since this would require maintaining information indefinitely and the amount of information to be maintained would grow indefinitely large. However, it is possible for an implementation to support the *waiting* and *executing* verbs, as we shall now discuss.

The "waiting" verb: In object models which provide only synchronous calls to objects, there is an upper limit on the number of pending requests (equal to the number of processes which the system supports). Similarly, in object models which support asynchronous calls, there are usually fixed size message buffers at either the client or server sides which limit the number of pending requests. Thus, in both synchronous and asynchronous systems, there is an upper limit on the number of pending requests and hence it is possible for an implementation to support the *waiting* verb.

The "executing" verb: An object operation may invoke itself recursively resulting in a single process *executing* the operation an unbounded number of times.[11] Thus there would appear to be no upper limit on the amount of information to store in order to support the *executing* verb. However, synchronised objects do not usually engage in deeply recursive invocations so, *in practice* there tends to be an upper limit on the amount of information which would need to be maintained.

[11]Subject, of course, to stack space limitations.

5 Evaluation of Power

Bloom [6] insists that an ideal synchronisation mechanism should exhibit several characteristics:

- A straightforward way to combine independent synchronisation constraints to construct more complex constraints. (Bloom calls this the *ease of use* requirement.)
- Separation of code to implement an object's operations from the code to provide synchronisation for the object. (This is a *modularity* requirement.)

The *ease of use* requirement is met by being able to combine constraints with the boolean operators **and**, **or**, **not** etc. This can be clearly seen in the solution to the Bounded Buffer (Figure 2) where the independent constraints "$exec(\text{Put}) = 0$" and "not_full" are combined without difficulty.

The **entry guards** clause separates synchronisation from the code implementing an object's operations; this meets the *modularity* requirement.

Bloom also lists six types of information which a synchronisation mechanism should have access to in order for it to have good expressive power. Most synchronisation mechanisms provide access to at least three of these types of information: (i) the operation requested, (ii) the synchronisation state of the object and (iii) history information. The power of scheduling predicates shows itself in not only providing access to the these types of information, but also in allowing the (iv) parameters and (v) relative *arrival* times of requests to be compared. Mediators [13] and Caromel [9] are two of the few other mechanisms which also provide this level of power.

Alas, scheduling predicates do not allow (vi) instance variables to be used *safely* in entry guards. This is a serious limitation since the solutions to many synchronisation and scheduling policies require access to these. The next section discusses this problem in more detail.

6 The Problem of Instance Variables

Synchronisation state (synchronisation counters and information about the set of pending requests etc.) is only updated under well defined circumstances: whenever an operation is *requested*, or has *started* or *terminated* execution. This makes it possible for the compiler to generate code which guarantees that:

1. Synchronisation state is updated atomically with respect to the evaluation of entry guards, i.e., guards may only be evaluated when synchronisation state is consistent
2. Entry guards will be re-evaluated whenever (relevant) synchronisation state has been modified

These two guarantees are necessary for a correct implementation.

The updating of instance variables is not as restrictive as that of synchronisation state. Instance variables may be updated by direct assignment, or indirectly through pointers or when passed as reference parameters. Thus, unlike synchronisation state, we can not easily predict when instance variables will be updated.

If instance variables are allowed to appear in entry guards then we need to provide the above two guarantees for instance variables as well as for synchronisation state, i.e., we need to guarantee that:

1. Entry guards may only be evaluated when instance variables, referenced in the guards, are in a consistent state
2. Entry guards will be re-evaluated whenever instance variables, referenced in the guards, have been modified

It is difficult to imagine how these guarantees could be met in an elegant manner. Although we have already done some research in this area, we have yet to find a construct which is totally suitable, and the problem remains open.

7 Unification of Concepts

This section discusses how scheduling predicates subsumes some other notable synchronisation mechanisms.

7.1 Synchronisation Counters

One can consider synchronisation counters to be a restricted form of scheduling predicates. The following equalities hold:

$$\begin{array}{l} wait(Op) \equiv count(\text{var\_id } waiting\ Op : \mathbf{True}) \\ req(Op) \equiv count(\text{var\_id } requested\ Op : \mathbf{True}) \\ start(Op) \equiv count(\text{var\_id } started\ Op : \mathbf{True}) \\ exec(Op) \equiv count(\text{var\_id } executing\ Op : \mathbf{True}) \\ term(Op) \equiv count(\text{var\_id } terminated\ Op : \mathbf{True}) \end{array}$$

The significance of this is that instead of having two complementary, but separate, mechanisms with which to solve synchronisation/scheduling problems, we now conceptually have a single generalised mechanism. Having a minimum of conceptual constructs makes it easier to reason about the behavior of a concurrent program.

7.2 The "By" Clause

Like synchronisation counters, the **by** clause is also a restricted form of scheduling predicates: "**by** *parameter*" is equivalent to:

$$there\_are\_no(\text{p } waiting \text{ Op: p.}parameter < my\_req.parameter).$$

Expressions are just as easy to handle as single parameters. The more general form of the **by** clause is:

$$\mathbf{by}\ p_1,\ p_2, \ldots, p_n$$

This maps into the following general scheduling predicate form:

$$there\_are\_no \left(\begin{array}{l} O \ waiting \ operation\_id : \\ \bigvee_{i=1}^{n} \left(\bigwedge_{j=1}^{i-1} O.p_j = my\_req.p_j \right) \textbf{ and } O.p_i < my\_req.p_i \end{array} \right)$$

In the above formula $\bigvee$ denotes the combining of its terms with the boolean operator **or**; similarly $\bigwedge$ is used to denote **and**. In this generalised form, this formula looks unwieldy. For a concrete example of this with $n = 2$, the reader is referred back to the Shortest Job Next scheduler with FIFO sub-ordering, as shown in Figure 4.

7.3 Path Expressions

It has been shown elsewhere [16] that Predicate Path Expressions [2] can be implemented in terms of synchronisation counters. Since we have already shown in Section 7.1 that synchronisation counters are simply a restricted form of scheduling predicates, it follows that Predicate Path Expressions are also subsumed by scheduling predicates.

8 Efficient Implementation

This section deals with the efficient implementation of scheduling predicates. It is organised as follows:

Section 8.1 discusses some compiler optimisation techniques. These techniques are independent of any particular object model. Section 8.2 then gives a brief overview of some run-time optimisation techniques suitable for passive objects in shared memory.

8.1 Compile time Optimisation

8.1.1 Re-evaluation Matrices

One method to improve run-time efficiency is for the compiler to examine the entry guards and construct a re-evaluation matrix; this matrix informs the run-time when it is necessary to re-evaluate each guard. Without this, the run-time would have to re-evaluate all guards whenever an operation call is requested, started or terminated.[12] Recall the entry guards used in the solution of the Bounded Buffer problem:

entry guards
#define not_full = *term*(Put) - *term*(Get) < size
#define not_empty = *term*(Put) - *term*(Get) > 0
Put: *exec*(Put) = 0 **and** not_full;
Get: *exec*(Get) = 0 **and** not_empty;

An initial re-evaluation matrix can be obtained by direct examination of these:[13]

[12]Guide, which uses synchronisation counters, only re-evaluates guards when an operation terminates. This can lead to process starvation [16] and deadlock [11].

[13]There is no need to have the counters *wait* and *exec* in the matrix since they are macros defined in terms of *req*, *start*, and *term*. Similarly, there is no need to have *arrival* in the matrix since this is updated whenever a *req* counter is incremented.

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

Each operation has one row in the matrix, giving the set of synchronisation counters that, when changed, could make the operation's guard become true. Thus we note that if a call to the *Put* operation is blocked because its entry guard evaluates to false then the entry guard need only be re-evaluated when there is a change in one of the counters: *start*(*Put*), *term*(*Put*), or *term*(*Get*).

An important optimisation is to remove some of the entries from the matrix, thereby avoiding more unnecessary re-evaluations. In the solution to the bounded buffer problem, the expression "$exec(Put) = 0$" leads to *Put*'s entry guard being re-evaluated when *start*(*Put*) is updated. However, a pending *Put* request need not have its guard re-evaluated whenever *start*(*Put*) is incremented since incrementing *start*(*Put*) can only make *exec*(*Put*) non-zero. Similarly, the entry guard of *Get* need not be re-evaluated when *start*(*Get*) is updated. Taking this into account, the re-evaluation matrix can be reduced to:

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

Three comments are in order about this optimisation on the matrix. Firstly, a similar optimisation can be applied to entry guards containing the expression "$wait(Put) = 0$". In this case, a request need not have its entry guard re-evaluated whenever *req*(*Put*) is incremented, since this 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 "$exec(Put) \leq 1$" was given then the entry guard *would* have to be re-evaluated when *start*(*Put*) is incremented. Nevertheless, the optimisation is worthwhile because the constant 0 is so frequently used. In fact, all but one of the example programs in this report have expressions of the form "$exec(Op_i) = 0$" or "$wait(Op_i) = 0$".

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

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

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

The re-evaluation matrix can also cater for entry guards containing scheduling predicates. An entry guard containing an expression of the form

there_are_no(p *waiting* Op_i: <boolean expression>)

needs to be re-evaluated whenever a change occurs to the set of pending Op_i requests, that is, whenever $req(Op_i)$ or $start(Op_i)$ are updated, indicating an addition to or removal from the set, respectively.

8.1.2 Transformations of Scheduling Predicates

Even allowing for these optimisations, it is likely that the evaluation of guards will be expensive relative to the cost of using a less powerful synchronisation mechanism. Therefore, an interesting research topic will be recognising that certain patterns of entry guards specify particular synchronisation policies, and then generating code which will implement these policies in a cheaper manner.[14]

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

> **entry guards**
> A: *exec*(A, B) = 0;
> B: *exec*(A, B) = 0;
> C: *exec*(C) = 0;

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

Transformations may also be applied to guards with scheduling predicates. For example, an intelligent compiler might recognise that the entry guard shown in Figure 4 (SJN with FIFO sub-ordering) queues requests based on the tuple (*len*, *arrival*) and hence generate code to maintain an ordered linked list of requests.

If implemented, such transformations would offer the programmer the high-level declarative power of scheduling predicates but with the efficiency of low-level procedural mechanisms.

8.2 Optimisation for Passive Objects in Shared Memory

In a passive object model, objects are held in shared memory which may be accessed by several processes simultaneously. In such a system there is a one-to-one mapping between pending requests and blocked processes. It thus seems natural for each blocked process to re-evaluate its own guard whenever a relevant (as determined by the re-evaluation matrix) synchronisation counter has been updated.[15] However, this has the disadvantage that if several processes need to re-evaluate their guards then many context switches will take place.

A better approach would be for the process which updated the synchronisation counter to re-evaluate the entry guards on behalf of the other blocked processes, and only wake them up if they evaluate to true. This removes the possibility of a process being woken up to re-evaluate its entry guard only to find that it is still false. Hence, unnecessary context switches are avoided.

[14]Optimisation by transformation has been applied successfully in other contexts: e.g., tuple space operations in Linda [20].

[15]In the current implementation of Guide, each process re-evaluates its own guard.

9 Synchronisation Counters Overflow

One problem with synchronisation counter based mechanisms is that, for long-lived objects, the counters will eventually overflow. A solution to this problem has yet to be found. This section briefly examines some ways in which the problem might be tackled.

Let us make the (unrealistic) assumption that in the next few years CPU speeds will increase dramatically but that their word size will remain at 32 bits. With this increase in speed, assume that we have a maximum invocation rate of 1 invocation on a given object every microsecond. At this rate, we can expect counter overflow in just over one hour.

One obvious way to alleviate the problem is to increase the size of the counters from 32 to, say, 64 bits. This results in counter overflow being postponed for approximately 500,000 years. If need be, 64 bit counters can be achieved on current machines, at a slight performance penalty, by performing multi-word arithmetic. When 64 bit CPUs appear in a few years, this extended lifetime will be gained for free.

Another obvious, but flawed, solution would be to reset all the synchronisation counters to 0 whenever the object is in a quiescent state. To see the flaw in this scheme, consider the bounded buffer (Figure 2). The number of elements currently in the buffer is given by the expression:

$$\mathit{term}(\text{Put}) - \mathit{term}(\text{Get})$$

This expression is used in the entry guards to determine if the buffer is empty or full. If the synchronisation counters were reset to zero then entry guards would lose count of the number of elements in the buffer and thus the buffer contents would be lost.

However, the idea of resetting counters does have potential. It might be possible to introduce a new language construct in which the programmer would specify conditions under which counters could be *safely* decremented. More ambitious would be for the compiler to determine these rules by itself, thus relieving the programmer of the responsibility.

Lastly, it is worth noting that if techniques are developed for optimisation by transformation (Section 8.1.2) then by eliminating synchronisation counters, the problem of their overflow disappears.

10 Conclusions and Future Work

This report has presented a new notation called *scheduling predicates*. Examples like the Shortest Job Next scheduler, Figure 4, and the FIFO readers/writer, Figure 5, exemplify their suitability as a *specification notation* for synchronisation/scheduling policies.

As well as being a specification notion, it is also feasible to implement scheduling predicates. The potential for optimisation, as discussed in Section 8, shows that their high expressive power need not result in slow execution speed.

An evaluation of power was given in Section 5; few other synchronisation mechanisms can claim a similar level of power, and of those that do, none are declarative. The main failing of scheduling predicates is that they cannot be used for policies which require access to the instance variables of an object.

We are currently investigating how scheduling predicates can be integrated with inheritance and also extended to allow instance variables. A prototype implementation is under way.

Acknowledgements

We would like to thank the other members of the research group whose constructive criticism has helped in the development of the synchronisation mechanism presented in this report. This research is partially funded by the ESPRIT Comandos project (numbers 834 and 2071).

References

[1] The Programming Language Ada Reference Manual. Published in: Lecture Notes in Computer Science, Vol 155, Springer-Verlag, 1983.

[2] Sten Andler. Predicate Path Expressions. In *Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 226–236, San Antonio, Texas, 1979.

[3] Françoise André, Daniel Herman, and Jean-Pierre Verjus. *Synchronisation of Parallel Programs.* Studies in Computer Science. North Oxford Academic, 1985. Original French language edition (Synchronisation de Programmes Parallèles, Dunod) ©BORDAS 1983.

[4] Gregory R. Andrews. Synchronising Resources. *ACM Transactions on Programming Languages and Systems*, 3(4):405–430, October 1981.

[5] 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, February 1990.

[6] Toby Bloom. Evaluating Synchronisation Mechanisms. In *Seventh International ACM Symposium on Operating System Principles*, pages 24–32, 1979.

[7] Per Brinch Hansen. Distributed Processes: A Concurrent Programming Concept. *Communications of the ACM*, 21(11):934–941, November 1978.

[8] 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.

[9] 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.

[10] 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):105–107, April 1989.

[11] D. Decouchant, P. le Dot, and M. Riveill. A Synchronisation Mechanism for an Object Oriented Distributed System. Bull-IMAG, Z. I. de Mayencin - 2, rue Vignate, 38610 Gières - France, February 1990.

[12] Stefano Genolini, Andrea Di Maio, Cinzia Cardigno, Stephen Goldsack, and Colin Atkinson. Specifying Synchronisation Constraints in a Concurrent Object-Oriented Language. In *Technology of O-O Languages and Systems (TOOLS '89).*

[13] 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.

[14] C.A.R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[15] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[16] Ciaran McHale. Pasm: A Language for Teaching Concurrency. B.A. project report, Department of Computer Science, Trinity College, Dublin 2, Ireland, April 1989.

[17] Christian Neusius. Synchronising Actions. In Pierre America, editor, *ECOOP '91*, pages 118–132, Geneva, Switzerland, July 1991. Springer-Verlag. Available as Volume 512 of *Lecture Notes in Computer Science.*

[18] O. M. Nierstrasz. Active Objects in Hybrid. In Norman Meyrowitz, editor, *OOPSLA '87 Proceedings.* ACM. Special issue of *ACM SIGPLAN Notices*, 22(12):243–253.

[19] David P. Reed and Rajendra K. Kanodia. Synchronisation with Eventcounts and Sequencers. *Communications of the ACM*, 22(2):115–123, February 1979.

[20] Steven Ericsson Zenith. Linda coordination language: subsystem kernel architecture (on transputers). Research Report YALEU/DCS/RR-794, Yale University, Departmemt of Computer Science, New Haven, Connecticut. USA., 29 May 1990.