Amadeus Project

Synchronisation Variables

Ciaran McHale, Seán Baker, Bridget Walsh, Alexis Donnelly
Distributed Systems Group
Department of Computer Science
University of Dublin
Trinity College, Dublin 2, Ireland.
Fax: +353-1-772204
Tel: +353-1-7021539
Email: {firstname.lastname}@cs.tcd.ie, bwalsh@iona.ie

Abstract

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 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.

In this paper, we study this problem in depth and solve it by defining a framework to guide the design of synchronisation mechanisms. In solving the problem, our framework illustrates that, contrary to popular belief, access to instance variables by synchronisation mechanisms is *not* required in order to implement synchronisation policies that apparently rely on the state of the object—such state can be maintained by the synchronisation code itself.

Our framework offers additional benefits. Synchronisation mechanisms designed within the guidelines of the framework can possess considerable expressive power. (The example synchronisation mechanism we present subsumes the power of numerous other synchronisation mechanisms.) Also, we show that most of the concepts of our framework can be implemented in terms of existing language constructs, thus minimising complexity that needs to be added to a sequential language in order to support concurrency.

Document Status Final version
Distribution Public
Document # Technical Report TCD-CS-94-01
This version January 19, 1994

Contents

Synchronisation Variables

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

Abstract

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 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.

In this paper, we study this problem in depth and solve it by defining a framework to guide the design of synchronisation mechanisms. In solving the problem, our framework illustrates that, contrary to popular belief, access to instance variables by synchronisation mechanisms is *not* required in order to implement synchronisation policies that apparently rely on the state of the object—such state can be maintained by the synchronisation code itself.

Our framework offers additional benefits. Synchronisation mechanisms designed within the guidelines of the framework can possess considerable expressive power. (The example synchronisation mechanism we present subsumes the power of numerous other synchronisation mechanisms.) Also, we show that most of the concepts of our framework can be implemented in terms of existing language constructs, thus minimising complexity that needs to be added to a sequential language in order to support concurrency.

1 Introduction

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.

[*]Distributed Systems Group, Department of Computer Science, Trinity College, Dublin 2, Ireland. Tel: +353-1-7021539. Email: {firstname.lastname}@cs.tcd.ie, bwalsh@iona.ie

Surprisingly, many recent language designs (e.g., Guide [DDR+90], PLOOC [Tho92], CEiffel [Löh92] 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 this paper, we study this problem in depth and solve it by defining a framework to guide the design of synchronisation mechanisms. A central concept in our framework is that variables required to implement the sequential functionality of an object should be classified as "(sequential) instance variables", and the variables required to implement an object's synchronisation policy as "synchronisation variables". We will show that, in practice, there is no overlap between these two sets of variables, i.e., the case does not arise in which a variable is used to implement *both* the sequential functionality *and* the synchronisation policy of an object. Thus, by providing language support for synchronisation variables, the problems associated with synchronisation code accessing instance variables disappears.

We will show that our framework offers additional benefits. Synchronisation mechanisms designed within the guidelines of our framework can possess considerable expressive power. Also, most of the concepts of our framework can be implemented in terms of existing language constructs, thus minimising the amount of complexity that needs to be added to a sequential language in order to support concurrency.

1.1 Structure of the Paper

The rest of this paper is structured as follows.

Section 1.2 explains some basic concepts and terminology used in this paper. Section 2 considers some previous attempts to allow instance variables to be used safely in synchronisation code, and discusses their limitations. In Section 3 we take a fresh look at the problem, and propose a framework to solve it. In Section 4 we illustrate this framework by applying it to an existing synchronisation mechanism and use the resultant mechanism to solve several synchronisation problems. We summarise the benefits of our approach in Section 5, and then, in Section 6, outline how the core concepts of our framework can be mapped onto existing constructs of an object-oriented language. We compare the core concepts of our synchronisation framework with related work in Section 7. Finally, in Section 8, we conclude the paper and suggest areas for future research.

1.2 Terminology and Concepts

In this paper, synchronisation constraints are represented as *guards*, i.e., conditions, associated with operations.[1] The principle of using guards is simple: whenever an invocation is attempted on an operation, it will be blocked until that operation's guard becomes true.

Synchronisation counters [RV77] often appear in guard-based mechanisms. These (automatically maintained) variables of an object count the total number of invocations for each operation of the object which have *arrived* at the object, have *started* execution and have *terminated* execution, etc. The two counters of most interest to this paper

[1]Guards are simply the syntactic notation which we have chosen to use in this paper. The principles we discuss apply to synchronisation mechanisms in general, whether they are guard-based or not.

are *exec* and *wait*. The counter "*exec*(Foo)" indicates how many invocations on operation *Foo* are currently *executing*, and "*wait*(Foo)" indicates how many invocations are currently *waiting* to execute *Foo* (they are waiting because their guards have not yet evaluated to true). In this paper, the form "$exec(A, B, \ldots, Z)$" is used as a shorthand for "$exec(A) + exec(B) + \cdots + exec(Z)$". As an example, the following guards can be used to enforce a basic readers/writer policy on an object which has operations named *Read* and *Write*:

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

Scheduling Predicates (SP) [MWBD91] provide a way for guards to compare the parameters, and relative arrival time of invocations, thus facilitating the implementation of more complex 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. 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 invocations.

We will finish this section by briefly distinguishing between declarative and procedural synchronisation mechanisms.

Many guard-based synchronisation 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. As a case in point, the FCFS example discussed previously is declarative in nature—it specifies the desired synchronisation policy without considering low-level, implementation details. While declarative mechanisms can often give elegant, concise solutions to some synchronisation problems, they are generally limited in power because they cannot express algorithms.

In contrast to declarative mechanisms, some mechanisms are *procedural*. These combine synchronisation primitives with sequential flow control constructs and data structures, allowing programmers to implement synchronisation policies through algorithms. In being able to utilise algorithms, procedural mechanisms generally have more expressive power than declarative ones. However, the synchronisation primitives provided are usually at a low-level and this often results in verbose and complex implementations of synchronisation policies.

2 Previous Attempts to Solve the Problem

In this section, we review some existing approaches to allowing instance variables to be used safely in guards.

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.

```
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 1: The Bounded Buffer

Consider the bounded buffer code in Figure 1. For $\mathcal{V} = num$, we note the following:

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

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

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.

```
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: First attempt at the Dynamic Priority Print Queue problem

A first attempt at implementing this is given in Figure 2. 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*(...)

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 The Guide Proposal

For some time, Guide [DDR+90] 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. 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 "$\mathrm{lock}_{\mathrm{sync}}$".
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 $\mathrm{lock}_{\mathrm{sync}}$;
   $\mathcal{V}$ := tmp;
   re-evaluate guards which rely on $\mathcal{V}$;
release $\mathrm{lock}_{\mathrm{sync}}$;
```

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 paper rather than in the syntax of Guide.)

Dynamic Priority Print Queue

Consider the first attempt at the dynamic priority printer shown previously in Figure 2. When we originally discussed this in Section 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])
local GroupId   index;
{
   for index in one..staff do
      priority[index] := NewPriority[index];
   end
}
```

If the code generated by the compiler acquires and releases $\mathrm{lock}_{\mathrm{sync}}$ *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 $\mathrm{lock}_{\mathrm{sync}}$ around the entire for-loop.

The compiler writer might be able to take into account this interaction of for-loops and the usage of $\mathrm{lock}_{\mathrm{sync}}$, 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$_{\text{sync}}$ would surrounded *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

```
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 3: First attempt at Guide solution to the Dining Philosophers problem

simulating the action 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 3 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 4. However, this solution works, not because it makes use of Guide's proposed semantics, but

```
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 4: Guide solution to the Dining Philosophers problem

rather because it falls back to using the technique discussed in Section 2.1. You can see this by noting that, for $\mathcal{V}$ = *chopstick_avail[]*, we have:

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

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 revert to the technique discussed in Section 2.1, i.e., reduce the potential for concurrency within objects.

The other languages mentioned in the introduction ignore the problem entirely.

3 A New Approach

In this section we take a fresh look at the problem, and propose a framework to solve it. To keep the concepts of this framework separate from the idiosyncrasies of any particular synchronisation mechanism to which it might be applied, we present the framework in this section and defer, until Section 4, presentation of a specific mechanism which utilises it.

Our complete synchronisation framework consists of the following four principles:

- The variables needed to synchronise an object are segregated from the variables needed to implement the object's sequential functionality. (This is discussed in Section 3.1.)
- The programming of synchronisation policies is treated as a form of event-based programming. (This is discussed in Section 3.2.)
- Synchronisation code has a mechanism by which it can cause a pending invocation to start executing. (This is discussed in Section 3.3.)
- Synchronisation code can access information about invocations. (This is discussed in Section 3.4.)

3.1 Synchronisation Variables

Three types of variable are indicated in Figure 5.

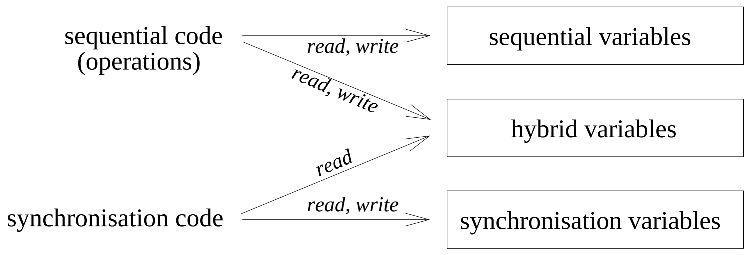

Figure 5: Graphical description of an object's code and variables

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 used in *both* the sequential code and the synchronisation code. Examples of hybrid variables in this paper so far have included: (i) *num* in the Bounded Buffer (Figure 1); (ii) *priority* in the Dynamic Priority Print Queue (Figure 2); and (iii) *chopstick_avail* in the Dining Philosophers (Figures 3 and 4).

It is these hybrid variables (rather than the purely sequential or purely synchronisation variables) which causes problems because, as we have seen in Section 2, it is difficult to ensure that the synchronisation code will always see them in a consistent 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, we propose that languages provide support for two categorisations of variables: sequential and synchronisation.

We defer, until Section 5.2, discussion on how to handle the hypothetical case of variables which are truly hybrid.

3.2 Synchronisation as Event-based Programming

Having argued for the existence of synchronisation variables, a question now arises of how they should be maintained. We think it would be unsuitable to maintain synchronisation variables in the bodies of operations for two reasons:

1. To aid modularity, synchronisation code should be kept separate from sequential code [Blo79].
2. Synchronisation code placed inside an operation's body could be executed only when the operation is executing. This is often inappropriate for synchronisation code. For example, sometimes it is desirable to update a synchronisation variable when an invocation *arrives* but *before* it has been allowed to start executing.

Instead, we propose an event-based approach in which programmers may specify *actions* to be executed at *events*. It is these actions which are used to update synchronisation variables. The events we consider are:

- The *arrival* of an invocation for an operation. For example, the event *arrival*(Read) indicates that an invocation upon the *Read* operation has arrived.
- The *start* of execution of an invocation. For example, the event *start*(Read) indicates that an invocation for the *Read* operation has started execution.
- The termination (hereafter abbreviated to *term*) of execution of an invocation. For example, the event *term*(Read) indicates that an invocation of *Read* has terminated execution.

We indicate that an action is to be executed at an event with the notation:

event → action

For example:

arrival(Bar) → foo := foo + 1;

This specifies that *foo*, a synchronisation variable, is to be incremented whenever the event *arrival*(Bar) occurs.

Note that the execution of actions take place in mutual exclusion with respect to other actions. This guarantees that actions will always see synchronisation variables in a consistent state.

3.3 Starting Invocations

All synchronisation mechanisms need to have a way to trigger the transition from *arrival* to *start* of execution for an invocation. There are several common ways in which synchronisation mechanisms may 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 has been illustrated by many of the examples so far in this paper. The guard, associated with an invocation, evaluating to true will trigger the *start* event for that invocation.

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

Note that if, say, guards, are provided as the means to cause invocations to start execution then the evaluation of guards and the execution of actions must take place in mutual exclusion with respect to other guards and actions. This guarantees that guards and actions always see synchronisation variables in a consistent state.

3.4 Access to Information about Invocations

A synchronisation mechanism needs to be able to access information about invocations of operations on an object. The level of access permitted varies greatly from one synchronisation mechanism to another, but without *any* access at all, the power of a synchronisation mechanism would be limited to such an extent as to make it virtually useless. For example, if a synchronisation mechanism could not access even the name of the invoked operation (a very basic piece of information) then the synchronisation mechanism would be incapable of placing different synchronisation constraints on different operations.

Aside from the name of the invoked operation, other potentially useful information associated with invocations includes the arrival time of the invocation (used in, e.g., a FCFS scheduler) and parameters of the invocation (e.g., a *length* parameter used in the Shortest Job Next scheduler).

Our framework makes the following accessibility requirements upon invocations:

- All the information about an invocation, that a synchronisation mechanism has access to, should be grouped together in some kind of data structure.
- A guard/action should have access to information about the *current* invocation, i.e., the invocation for which it is being evaluated/executed.
- The run-time should automatically maintain information about invocations in a list/collection so that synchronisation code can iterate over that list/collection 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 that have terminated execution.

It is outside the scope of the framework 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; or the language construct used to iterate over such collections. These details will vary from one language to another. However, we illustrate the principles with 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[2] 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.

Aside from the accessibility requirements already discussed, our framework makes two other requirements upon invocations:

- Parameters of an invocation must not be accessed by synchronisation code while they are being updated by sequential code. (This is discussed in Section 3.4.1.)
- It should be possible for programmers to declare synchronisation variables local to invocations. (This is discussed in Section 3.4.2.)

In making these two final requirements, we achieve the symmetry of sequential and synchronisation variables shown in Table 1. Thus, we argue, our introduction of "synchronisation variables" has not added new concepts to language design, but rather has generalised the existing concept of "variables".

[2]Our framework does not specify what syntax should be used to declare synchronisation variables as this will vary to suit the host language. Some example syntax for declaring synchronisation variables can be found in Section 4.2.

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

Table 1: Symmetry of sequential and synchronisation variables

3.4.1 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. 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), and the table position, *pos*, passed to *Eat* in the Dining Philosophers problem (Figures 3 and 4).

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).

A technique is required to ensure that hybrid parameters are not accessed by synchronisation code while they are being updated by sequential code. One way to achieve this is to arrange for the parameters of an invocation to be copied when that invocation arrives, i.e., at the *arrival* event. This *copy* can be accessed by the synchronisation code, safe in the knowledge that if the body of an operation updates a parameter then this update will not affect the synchronisation code's copy of that parameter, and vice versa. A simple optimisation is for the runtime to copy only those parameters which are used in synchronisation code.

3.4.2 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 by 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.

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.[3]

4 An Example Mechanism

We now illustrate the concept of synchronisation variables, introduced in Section 3, by adding them to the Scheduling Predicates [MWBD91] synchronisation mechanism. We call the resultant mechanism Esp (an Extension of Scheduling Predicates).

4.1 Notation

Figure 6 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.

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

Figure 6: Layout of a class

As mentioned in Section 3.2, 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 into synchronisation operations (line 7) which the actions can invoke.

4.2 Examples

In this section, we illustrate the usage of synchronisation variables through examples. These examples are organised into three groupings. The first group of examples illustrate how

[3]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: (i) it moves the inconvenience of calculating job length from clients to the service object, thus making the service more convenient to use; and (ii) it prevents clients from thwarting the server object's scheduling policy by under-estimating the length of their own jobs.

several synchronisation mechanisms are subsumed by our framework. The second group illustrates how our framework allows complex scheduling policies to be implemented easily. Finally, the last group is of examples 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

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

Synchronisation Counters

The code in Figure 7 declares three synchronisation variables—a, b and c. These are all initialised to zero when an object is created. (In the notation of this paper, the constructor for 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 for 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 {
  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 7: 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, ESP provides synchronisation counters 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 8) 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 variable 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 {
  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 8: 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 9 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 our framework subsumes Scheduling Predicates. As with synchronisation counters and *arr_time*, ESP provides scheduling predicates as a form of syntactic sugar.

4.2.2 Scheduling Power

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

```
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 9: First-come, first-served Printer

Alarm Clock

In the Alarm Clock problem [Hoa74], an operation, *Sleep*, must be implemented which will delay for a specified *period* of time. An usual assumption is that it is possible to arrange for an operation, *Tick*, to be invoked periodically (say, once a second) to mark the passage of time.

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

Figure 10: The Alarm Clock

Our solution is shown in Figure 10. In this, the counter *term*(Tick) is used to indicate the current time. We associate a variable, *wakeup_time*, with each *Sleep* invocation and calculate its value at the *arrival*(Sleep) event. The resultant guard on *Sleep* is trivial and intuitive.

Notice that the operations *Sleep* 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.

```
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 11: Starvation-free, Shortest Job Next scheduler

Figure 11 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 paramter 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*.[4]

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

Dining Philosophers (Starvation-free Version)

The Guide solution to the Dining Philosophers problem shown in Figure 4 does not guarantee against 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 12 illustrates this. This solution makes use of two predicates in the guard of *Eat*.

[4]This requires, as suggested in Section 3.4.1, that two copies of parameters 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.

```
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 12: Starvation-free solution to the Dining Philosophers problem

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—an iterator determines which invocations were skipped over, and increments their *skipped* values.

4.2.3 Instance Variables

In this paper we have 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 back up this claim by taking several synchronisation problems from the literature that in the past have been implemented via instance variables, and re-implementing them using synchronisation variables instead.

Dynamic Priority Print Queue

Earlier, in Section 2.1, we introduced the Dynamic Priority Printer and discussed the problems associated with trying to maintain the group priorities as instance variables.

Figure 13 shows an ESP implementation of this scheduler. The main difference between this and the previous attempt (Figure 2) is that the *priority[]* variable, and the code to maintain it, has 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 *Sleep* and *Tick* operations in the Alarm Clock) it is, in effect, just a hook into the synchronisation code.

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

class Printer {
   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 13: Solution to the Dynamic Priority Print Queue

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 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, it is reasoned, 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, thus making it a hybrid variable, as previously discussed in Section 2.1.

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.

```
class Buffer[elem, Size] {
  int    get_index, put_index;              elem    data[Size];
  Buffer() { get_index := 0;    get_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 14: Esp solution to the Bounded Buffer

The implementation of the bounded buffer shown in Figure 14 does not explicitly declare *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 12 showed an implementation of the Dining Philosophers that made use of predicates. Now we we show an alternative solution—one that makes use of synchronisation variables of the object rather than relying on precidates.

The code in Figure 15 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 4).

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

```
type TablePositon is int subrange(0..4);
class DiningPhilTable {
   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 15: Solution to the Dining Philosophers problem

are no more and then reverse direction [Hoa74]. This is sometimes called the "elevator algorithm" because it mimics the behaviour of a lift.

The code in Figure 16 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 16 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
```

(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.

```
const MaxCylinder = 100;
type CylinderNum is int subrange(0..MaxCylinder);
class DiskHeadScheduler{
  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 16: First solution to the Disk Head Scheduler

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 therafter. It decreases whenever any other invocation is serviced and it is possible to calculate how much it decreases by. 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 17 illustrates the changes in code needed to achieve this.

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. In a previous paper [MWBD91], we mention some compile-time optimisations for SP; some of these are also suitable for ESP. One of particular relevance to the current example is optimisation by transformation [MWBD91, pg. 190]. 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 need only choose the invocation at the head of the list.

The guard for *Transfer* in Figure 17 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.

	insert item	remove item	total cost
Original code (Figure 16)	O(1)	O(N^2)	O(N^2)
Optimised code (Figure 17)	O(N)	O(1)	O(N)

Table 2: Cost of maintaining the invocation list for the Disk Head Scheduler

Table 2 compares the run-time cost of our two versions of the Disk Head Scheduler. In our first solution (Figure 16), the run-time 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.

```
const MaxCylinder = 100;
type CylinderNum is int subrange(0..MaxCylinder);
class DiskHeadScheduler{
  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 17: Optimised solution to the Disk Head Scheduler

5 Discussion

Later in this paper we compare our work to that of others. But first, we summarise some of the benefits that our framework offers and tie up some loose ends.

5.1 Benefits of our Framework

The framework that we presented in this paper offers several important benefits.

5.1.1 No Conflict with Instance Variables

Searching the literature for synchronisation problems that *apparently* require access to instance variables in order to be solved brings to light problems such as the Bounded Buffer, the Dining Philosophers and the Disk Head Scheduler. 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, that, in practice, there is no overlap between an object's instance variables and its synchronisation variables.

5.1.2 Expressive Power

By generalising the concept of synchronisation variables to include parameters and variables local to operations, we gain considerable expressive power. We have illustrated this through the examples in Section 4.2.2 by implementing complex scheduling policies, e.g., the Disk Head Scheduler and starvation-free versions of the Dining Philosophers and the Shortest Job Next scheduler.

A better measure of power is that of Bloom's list of six types of information [Blo79] to which, she argues, synchronisation mechanisms should have access in order to have good expressive power. ESP provides access to all of these six types of information, with the exception that ESP accesses synchronisation variables rather than instance variables. Very few other synchronisation mechanisms have access to all six types of information.

5.1.3 Subsumption of Other Synchronisation Constructs

The power of our framework, as illustrated by ESP, is based on just four basic concepts: synchronisation variables; event-based programming; a construct that can cause a pending invocation to start executing; and information about invocations.

In fact, not one of these four concepts is new.

- Variables are ubiquitous in sequential programming languages, and automatically maintained variables (e.g., synchronisation counters and the arrival time of invocations) are not uncommon in synchronisation mechanisms either.
- 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, though few mechansisms allow programmers to associate actions with them.
- 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, to fulfill this purpose.

  The concept of guards—the construct employed in ESP—is well-know in sequential programming, e.g., Dijkstra's guarded commands [Dij75] and Eiffel's pre- and post-conditions [Mey92]. Guards also form the basis of several synchronisation mechanisms.
- Information about an invocation—notably its parameters and local variables—is accessible within the body of the invoked operation. Often such information is collectively referred to as an *activation record*. Our framework, in effect, makes similar information available to synchronisation code.

In embracing just these four concepts, ESP obtains a considerable amount of expressive power and actually subsumes several other synchronisation constructs such as synchronisation counters, SP and automatically maintained arrival times of invocations as shown in Section 4.2.1. It has been shown elsewhere [MWBD91, pg. 187–188] that Path Expressions and the "by" clause of SR can be implemented in terms of synchronisation counters and SP, respectively; since ESP subsumes the latter two, it follows that it also subsumes Path Expressions and SR's "by" clause.

5.1.4 Modularity

In splitting the variables of an object into sequential ("instance") variables and synchronisation variables, we have removed the dependence of the synchronisation code on the instance variables of an object. This on its own is not enough to ensure modularity since if a synchronisation mechanism has limited expressive power then it might need to mix synchronisation code with sequential code in order to implement some synchronisation policies. (An example of this is "synchronisation procedures" used in Path Expressions [Blo79, pg. 28].) However, independence of synchronisation code from instance variables *combined with* a high degree of expressive power makes it possible for Esp to fully segregate synchronisation code from sequential code without placing any apparent limits on the class of synchronisation problems which can be implemented.

We feel that keeping sequential code and synchronisation code segregated will improve the readability of classes. In general, readability is very subjective and so it is difficult to prove this claim; but, whatever about the readability of synchronisation code, it is surely easier to understand the sequential code of an operation if it does *not* have synchronisation code embedded in it.

We also feel that modularity will help with the inheritance and reuse of synchronisation code. We discuss this briefly in Section 6.5 but, unfortunately, space limitations prevent us from discussing this issue at length; the reader is directed to another paper [MWBD92] for a discussion of preliminary results in this area.

5.1.5 Declarative and Procedural Programming Styles

Esp maintains some synchronisation variables automatically (synchronisation counters and the *arr_time* of invocations, and it also makes copies of parameters accessed by the synchronisation code). If programmers use only guards and these automatically maintained synchronisation variables then they are using Esp in a purely declarative manner. On the other hand, if programmers make use of events and actions to maintain extra synchronisation variables, Esp appears more procedural in nature. As more and more use is made of events and actions, the more procedural (and, hence, less declarative) Esp appears to be. In essence, Esp offers programmers a spectrum of programming styles, from purely declarative to mainly[5] procedural, within a single framework.

The only other synchronisation mechanism we know of that offers both 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 procedural mechanism *or* the declarative mechanism, but not a mixture of both.

5.2 Hybrid Variables

In Section 3.1 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 our framework.

[5]It can be argued that Esp can not offer a programming style that is *fully* procedural in nature since a synchronised class will always contain at least one guard.

For a variable to be "hybrid" means that it is needed for the sequential functionality of an object and also 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.

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

Figure 18: Maintainence of sequential and synchronisation variables in step

Code to maintain the two variables in step is shown in Figure 18. 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.

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, in 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 point of view and insists that *num* is needed by both the sequential and synchronisation code then it is possible to accommodate this by use of the coding style shown in Figure 18: variable x corresponds to *num*, and operations A and B to *Put* and *Get*, respectively.

5.2.1 A Limitation of our Model

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 18 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.

```
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 19: Maintaining sequential and synchronisation variables in step under complex or non-deterministic logic

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 19.

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 hyphotetical curiosity rather than of practical concern.

6 An Implementation of Esp on top of an Object-oriented Language

We have recently completed a prototype implementation of Esp on top of the Dee language [Gro90]. In this section we briefly discuss how we have mapped the various concepts of our framework onto constructs of the Dee language.

6.1 A Brief Overview of Dee

Dee is a object-oriented language developed by its author 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, and booleans and strings. Dee supports generics—arrays, lists, sets and so on are provided as generic classes.

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

Dee was originally written to run on MSDOS. However, we have ported Dee to UNIX and modified the compiler to accept ESP. We have called the combination of Dee and ESP "DESP." Like Dee, DESP produces pcode.

6.2 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 *current* invocation.

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; (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 has important ramifications. Added complexity in the host language is avoided because several constructs—operations, actions and guards—are merged into one. This in turn makes it possible to inherit actions and guards just like normal operations which helps to overcome the conflict between inheritance and synchronisation.

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 6.3. For the moment, accept that invocations are represented at the language level by a type called Invocation.

The code in Figure 20 provides an example of the declaration of an action and a guard. 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 20: Declaration of actions and guards in DESP

The run-time 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 it takes as its name 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.

6.3 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 interchangable 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 is discussed in Section 6.3.1.)
- Represent invocation as a generic class that is instantiated upon operations. (This is discussed in Section 6.3.2.)

- Have an invocation class in which parameters are stored in a untyped list. (This is discussed in Section 6.3.3.)
- Use an inheritance hierarchy to model different invocation types. (This is discussed in Section 6.3.4.)

6.3.1 The Variant-record Approach

In a non object-oriented language, the different "invocation" types of a resource could all be combined into a single variant-record type. 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".)

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 reduce to one the number of tag fields 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. However, this would then means that "invocation" would not be a first-class type and as such there would almost certainly 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.[6]

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.

6.3.2 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*.

[6]Our argument for that variant records are not object-oriented is as follows. If one considers classes to be the object-oriented replacement of record types, then class hierarchies are the object-oriented replacement of variant records.

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 semantics 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.

6.3.3 The Untyped List Approach

Invocations could be represented by a class such as that shown in Figure 21. This approach 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).

```
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 21: Invocation class that stores parameters as a 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.

6.3.4 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.

A base class, *Invocation*, is shown in Figure 22. This class includes some instance variables that programmers may find useful for synchronisation.[7] These instance variables

[7]Previous examples in this paper 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.

will be initialised by the run-time 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 22: Base *Invocation* class in DESP

The base class, *Invocation*, does not include any parameters because, as we 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 previously shown in Figure 10. The synchronisation code accesses the *period* parameter of operation *Sleep*. The synchronisation code also declares a variable, *wakeup_time*, local to invocations on *Sleep*. In translating this ESP code to DESP, we create a subclass of *Invocation* as shown in Figure 23. This subclass contains *period* and *wakeup_time* as instance variables. The subclass also contains an operation to permit *wakeup_time* to be initialised.

```
class SleepInvocation
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 23: The *SleepInvocation* class

With the *SleepInvocation* class written, we can now write the code for the *AlarmClock* class. This class, shown in Figure 24, is a direct translation from the ESP version (Figure 10). Note that the guard and *arrival* action of *Sleep* take a parameter of type

SleepInvocation. The compiler will note that the *period* parameter of *Sleep* has a namesake in an instance variable of *SleepInvocation* and will generate code to copy this parameter at run-time.

```
class AlarmBlock
inherits Any

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

private method a_Sleep(t: SleepInvocation)
begin
  t.set_wakeup_time( t.period + term(Tick) ) end

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

map arrival(Sleep) → a_Sleep
      guard(Sleep) → g_Sleep
```

Figure 24: Desp implementation of an Alarm Clock

Discussion

Unlike the variant record approach (Section 6.3.1) and the technique of accessing parameters by their position in an untyped list (Section 6.3.3), 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 6.3.2).

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 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, at the time of writing, we have not introduced such a construct to DESP.

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 tiny minority 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 25) 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 philisophers.)

```
class SizeInvocation
inherits Invocation

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

Figure 25: The *SizeInvocation* class

6.4 New Language Constructs

The *waiting* and *executing* constructs of ESP were added to the language as builtin 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.

6.5 Inheritance of Synchronisation in Desp

Aside from being invoked automatically by the run-time, actions and guards in Desp are the same as "normal" operations. This means that actions and guards can be inherited in exactly the same manner as sequential operations. As the examples in Section 4.2 have shown, synchronisation policies in our framework tend to be implemented in terms of multiple guards and actions. Thus in Desp, synchronisation code will usually consist of several operations. Expressing synchronisation code in this fragmented form means that if a subclass modifies the synchronisation code inherited from its parent then it is likely to be able to reuse at least part of the parent's synchronisation code. This is in contrast to synchronisation mechanisms that express constraints as a single monolythic unit, e.g., Eiffel|| [Car90a]. In such mechanisms, any change to the synchronisation code in a subclass will require a complete rewrite of the synchonisation code.

Due to space limitations, we cannot discuss inheritance issues here any further. We leave an indepth discussion of this topic for a future paper.

7 Related Work

As we said earlier, our framework is composed of four concepts: (i) segregation of synchronisation variables from sequential variables; (ii) event-based programming; (iii) a way to cause invocations to start execution; and (iv) access to information about invocations. In this section we explore how well these concepts are supported in other synchronisation mechanisms.

7.1 Synchronisation Variables

As discussed in Sections 1 and 2, of this paper, many synchronisation mechanisms do not provide programmers with the ability to declare synchronisation variables, instead permitting them to access instance variables. We have shown that this access is usually either unsafe or at the expense of a reduction of the potential for concurreny within objects.

Aside from Esp, at least two other mechanisms—Mediators [GC86] and Synchronising Actions (SA) [Neu91]—propose a complete segregation of synchronisation code and variables from sequential code and variables. However, in papers on these mechanisms, the authors take it as read that modularity is desirable and do not mention that without it there would be problems with synchronisation code accessing instance variables. So although these mechanisms do have synchronisation variables, this is a result of modularity proposed for reasons different to our own. (Mediators is concerned with modular, concurrent programming in general while SA proposes segregation as an aid to overcome the conflict beteen synchronisation and inheritance.)

The SA support for synchronisation variables is only partial: while it allows synchronisation variables of an object, it does not allow synchronisation local variables or parameters.

7.2 Event-based Programming

Some influences of event-based programming can be found in many synchronisation mechanisms. For example, in the set-based synchronisation in ACT++ [KL89] and Rosette [TS89] state transitions (from one "enabled-set" to another) can occur when processing operation invocations—this is a form of event-based programming. Also, as demonstrated by the code in Figure 7, synchronisation counters are simply counts of how often particular events have occurred. Even the concept of guards is tied to events since they are evaluated *at* events.

Our framework embraces events quite explicitly, allowing progarammers to associate actions with each of the three types of events—*arrival*, *start* and *term*. The examples in Section 4.2 illustrate the utility of this. Most other synchronisation mechanisms do not allow programmers to associate actions with these events and this limits their expressive power.

The few synchronisation mechanisms which *do* permit programmers to associate actions with events include PLOOC [Tho92], Synchronising Actions (SA) [And81] and Mediators [GC86]; however, the first two of these mechanisms do not allow programmers to associate actions with all of the events and thus their expressive power is still limited.

7.3 Starting Invocations

As we said in Section 3.3, there are several ways for synchronisation mechanisms to cause invocations to start executing, guards being just one such way.

Not all synchronisation mechanisms that use guards offer safe re-evaluation semantics for guards, as we discuss in Section 7.3.1. Then, in Section 7.3.2 we discuss a benefit gained by keeping guards—or whatever construct a synchronisation mechanism employs to cause invocations to start executing—syntactically separate from the code of actions.

7.3.1 Guard Re-evaluation Semantics

Consider an invocation that arrives at an object and has its guard evaluated. If the guard evaluates to false then the invocation will be blocked until its guard becomes true. This raises the question of how often guards for blocked invocations should be re-evaluated. For intuitive guard semantics it is necessary to re-evaluate a guard whenever something occurs that might cause that guard to evaluate to true.

Since, in ESP, guards are expressed in terms of synchronisation variables, it follows that a guard must potentially be re-evaluated whenever a synchronisation variable has been updated; and since synchronisation variables can be updated only at events, it follows that re-evaluating guards at *arrival*, *start* and *term* events is sufficient to provide intuitive guard re-evaluation semantics.

A potential criticism of this scheme, however, is that frequent re-evaluation of guards might be inefficient, especially if guard evaluation is expensive. This is especially true in, say, the Guide system where each guard evaluation involves a context switch. Guide initially tackled this inefficiency by redefining the semantics of guard re-evaluation [DDR$^+$90]: rather than re-evaluate guards at all events, they were re-evaluated only at *term* events. However, this was a dangerous optimisation which could lead to process starvation [McH89, pg. 77–78] and deadlock [DDR+90, pg. 7]. Feedback from users of the Guide compiler has convinced the implementors to revert back to the safer semantics of re-evaluating guards at every event. Other systems which offer unsafe guard re-evaluation semantics include CEiffel [Löh91, pg. 16] and DRAGOON[Atk90, pg. 123-124]

There is actually no need to trade-off safe guard re-evaluation semantics for run-time efficiency since compile-time optimisation can minimise how often guards need to be re-evaluated [MWBD91, pg. 188–189].

7.3.2 A Benefit of Syntactic Separation

Consider the following three actions:

$arrival$(Foo) $\rightarrow$ bar := bar + 1
$start$(Foo) $\rightarrow$ bar := bar + 1
$term$(Foo) $\rightarrow$ bar := bar + 1

The only thing that distinguishes any one of these actions from the other two is the event (*arrival*, *start* or *term*) at which the action is executed. Thus, we see that ESP denotes actions in a consistent manner. This consistency in the denotation of actions is due to one thing: ESP's mechanism for causing an invocation to start executing, i.e., its guard, is kept *syntactically separate* from actions.

Some synchronisation mechanisms do *not* syntactically separate actions from the construct they use to cause invocations to start executing. The result is that such synchronisation mechanisms do not have a consistent notation for denoting actions.

For example, in Mediators an "exec" or "spawn" statement is used to cause an invocation to start execution. (The reason two statements are provided is that "exec" services the invocation synchronously while "spawn" forks off a process to service the invocation asynchronously.) The "exec" or "spawn" statement is syntactically placed inside (the Mediators' equivalent of) an *arrival* action. The result of this syntactic merging of *arrival* actions with the mechanism to cause invocations to start execution is that actions at different events are denoted differently. *Arrival* actions are explicitly denoted by a keyword, but *start* actions are denonted only by the fact that the code preceeds an "exec" or "spawn" statement. Furthermore, a *term* action is denoted by its own keyword *if* the invocation was serviced asynchronously (by the "spawn" statement); otherwise it is denoted only by the fact that its code is placed immediately after the "exec" statement.

We feel that such inconsistency in notation reduces the clarity of Mediators code.

7.4 Accessibility of Invocations

While it is becoming more common for synchronisation mechanisms to be able to access a lot of information about invocations—e.g., the name of the operation being invoked, the arrival time of the invocation, parameters etc.—very few synchronisation mechanisms acknowledge the concept of an invocation data-type that binds such information together in a single structure.

Of those mechanisms that *do* recognise the concept of invocations, not all regard them as first class objects. Indeed our own synchronisation framework (presented in Section 3) requires that invocations be somehow *accessible*, but *no* requirement is made that they be recognised as *language-level* data types. The omission of such a requirement from the framework was deliberate: we wanted to present our synchronisation framework in a language-independent manner. Our presentation of ESP also neglected to indicate a representation for invocations (a brief look at the examples in Section 4.2 will reveal that *this_inv* was never explicitly declared). It was not until we discussed the implemention of ESP on top of Dee (in Section 6) that we discussed how invocations might be treated as a language-level data type.

In Section 6.3 we outlined four possible ways in which invocations might be treated as language-level types.

Aside from DESP, we are not aware of any language that uses inheritance to model different invocation types.

Similarly, we do not know of any language that employs the use of a generic invocation class.

The untyped-list approach is more popular, being used in at least two synchronisation mechanisms built on top of Eiffel [Car90a, KB93]. As we said in Section 6.3.3, the main drawback of this approach is that accessing parameters by their position in an untyped list is error-prone and prevents the possibility of compile-time type-checking.

The variant record approach (Section 6.3.1) is also somewhat popular, being used in both Mediators and CEiffel.

In Mediators, the "job descriptor" (the Mediators term for what we call an "invocation") is stated to be:

> "... a variant record containing fields for a *key* variable [a unique handle for each invocation], the name of the *service* [i.e., operation] requested and the parameters for that service. The service field serves as a tag for variant parameter fields" [GC86, pg. 472].

(Annotations and emphasis added by authors.) In all the examples in the Mediators paper, variables for accessing "job descriptors" are implicitly declared within constructs of the synchronisation mechanism. This lack of an example of a explicit declaration of a "job descriptor" variable suggests that it is not a first-class data type, but rather that "job descriptor" is a pseudo-type type limited to appearing only within specific Mediator constructs. The authors of the Mediators paper do not discuss possible adverse effects that this idiosyncrasy may have on the integration of Mediators with other language concepts, but it appears to have one benign side-effect: limiting job descriptors to appear only within specific constructs makes it possible for the compiler to check the type-safety of access to these variant records.

CEiffel has a "predefined identifier," `Request`, [Löh91, pg. 17] which is stated to be "akin to a variant record type". `Request` is *not* a first-class data type since:

> `Request` denotes the type of the requests associated with [a *particular*] class. Data of a `Request` type *cannot* be passed between objects [Löh91, pg. 17].

(Emphasis added by authors.)

8 Conclusions

We started this paper by illustrating how permitting instance variables to be accessed by synchronisation mechanisms introduces a problem; one which, surprisingly, has gone unnoticed by many researchers over the years.

Having examined some previous attempts to address the issue and found them inadequate, we addressed the problem ourselves by defining a framework to guide the design of synhronisation mechanisms.

Our framework has several benefits.

Firstly, our framework solves the problem introduced at the start of the paper. In doing so it illustrates that, contrary to popular belief, access to instance variables by synchronisation mechanisms is *not* required in order to implement synchronisation policies which apparently rely on the state of the object—such state can be maintained by the synchronisation code itself.

Secondly, synchronisation mechanisms designed within the guidelines of the framework can possess considerable expressive power. We illustrated this by showing how an example mechanism, ESP, could implement many complex scheduling policies and, in fact, subsume the expressive power of numerous synchronisation constructs (synchronisation counters, automatically maintained *arrival times* of invocations, scheduling predicates such as *there_is_no*, the "by" clause of SR and Path Expressions).

Thirdly, most of the concepts of our framework can be implemented in terms of *existing* language constructs, thus minimising the amount of complexity that needs to be added to a language in order to support concurrency.

Also the framework's strict segregation of synchronisation code from sequential code ensures modularity. While it is often worthwhile to strive towards program modularity, we feel that the modularity we have obtained in DESP will be of particular help in tackling the conflict between synchronisation and inheritance. However, we leave discussion of this issue for a future paper.

Acknowledgements

We would like to thank Ann Barry, Alan Judge and Antony Courtney for comments on a previous version of this paper. This research has been partially funded by the Comandos ESPRIT project.

References

[And81] Gregory R. Andrews. Synchronising Resources. *ACM Transactions on Programming Languages and Systems*, 3(4):405–430, October 1981.

[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.

[BH78] Per Brinch Hansen. Distributed Processes: A Concurrent Programming Concept. *Communications of the ACM*, 21(11):934–941, November 1978.

[Blo79] Toby Bloom. Evaluating Synchronisation Mechanisms. In *Seventh International ACM Symposium on Operating System Principles*, pages 24–32, 1979.

[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.

[DDR+90] 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 Computer Systems*, Arlington, Texas, February 1990.

[Dij75] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[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.

[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.

[Hoa74] C.A.R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[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-Orientated Languages. In Stephen Cook, editor, *ECOOP 89*, pages 131–145. Cambridge University Press, July 1989.

[Löh91] Klaus-Peter Löhr. Concurrency Annotations and Reusability. Report B-91-13, Frachbereich Mathematik, Freie Universität, Berlin, November 1991. email: lohr@inf.fu-berlin.de.

[Löh92] Klaus-Peter Löhr. Concurrency Annotations. In *OOPSLA '92*, pages 327–340, 1992.

[LSAS77] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction Mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[McH89] Ciaran McHale. Pasm: A Language for Teaching Concurrency. B.A. project report, Department of Computer Science, Trinity College, Dublin 2, Ireland, April 1989.

[Mey92] Bertrand Meyer. *Eiffel: The Language.* Prentice Hall, 1992. ISBN 0-13-247925-7.

[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.*

[Neu91] 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.*

[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.

[Riv92] Michel Riveill. An Overview of the Guide Language. Presented at the OOPSLA '92 workshop on Objects in Large Distributed Applications, 1992. email: riveill@imag.fr.

[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.

[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.

[TS89] Chris Tomlinson and Vineet Singh. Inheritance and Synchronisation with Enabled-Sets. In *OOPSLA '89 Proceedings*, pages 103–112, October 1989.