Prioritised Choice over Multiway Synchronisation

Douglas N. WARREN [a,1],
[a] *School of Computing, University of Kent, Canterbury, UK*

Abstract. Previous algorithms for resolving choice over multiway synchronisations have been incompatible with the notion of priority. This paper discusses some of the problems resulting from this limitation and offers a subtle expansion of the definition of priority to make choice meaningful when multiway events are involved. Presented in this paper is a prototype extension to the JCSP library that enables prioritised choice over multiway synchronisations and which is compatible with existing JCSP Guards. Also discussed are some of the practical applications for this algorithm as well as its comparative performance.

Keywords. CSP, JCSP, priority, choice, multiway synchronisation, altable barriers.

Introduction

CSP [1,2] has always been capable of expressing external choice over multiway synchronisation: the notion of more than one process being able to exercise choice over a set of shared events, such that all processes making that choice select the same events. For some time, algorithms for resolving such choices at run-time were unavailable and, when such algorithms were proposed, they were non-trivial [3,4].

Conversely priority, a notion not expressed in standard CSP, has been a part of CSP based languages from a very early stage. Priority, loosely, is the notion that a process may reliably select one event over another when both are available. Whilst being compatible with simple events such as channel inputs, algorithms for resolving choice over multiway synchronisation have been incompatible with priority.

This paper introduces an algorithm for implementing *Prioritised Choice over Multiway Synchronisation* (*PCOMS*) in JCSP [5,6] through the use of the *AltableBarrier* class. This addition to the JCSP library, allows entire process networks to be atomically paused or terminated by alting over such barriers. They also enable the suspension of sub-networks of processes, for the purposes of process mobility, with manageable performance overheads. This paper assumes some knowledge of both JCSP and occam-π [7,8] – the latter being the basis of most pseudo-code throughout the paper.

This paper intends to establish that using the *AltableBarrier* class simplifies certain problems of multiway synchronisation. However, there are no immediately obvious problems which require *PCOMS* per se. For example, graceful termination of process networks can be achieved using conventional channel communication. However, if such networks have a complicated layout or if consistency is required at the time of termination then graceful termination using channel communication becomes more complicated. This same problem using *PCOMS* requires only that all affected processes are enrolled on (and regularly ALT over) an *AltableBarrier* which they prioritise over other events.

[1]Corresponding Author: *Douglas N. Warren*. E-mail: dnw3@kent.ac.uk.

Introduced in this paper are the background and limitations of existing multiway synchronisation algorithms. In Section 3 the limitations of existing notions of priority and readiness are discussed and proposals for processes to pre-assert their readiness to synchronise on barriers are made. This section also proposes the notion of nested priority, the idea that several events may be considered to be of the same priority but to exist in a wider priority ordering.

Section 5 details the interface that JCSP programmers need to use in order to include AltableBarriers in their programs. Section 6 details the inner workings of the algorithm itself. Section 7 details stress tests performed on the algorithm as well as comparative performance tests with the previous (unprioritisable) *AltingBarrier* algorithm. The results are discussed in Section 8 as well as some proposed patterns for implementing *fair alting* and for influencing the probability of certain events being selected through *partial priority*. Section 9 concludes the paper.

1. Background

This section considers some of the existing algorithms for resolving choice over multiway synchronisation both where the set of events are limited and where the set of events may be arbitrarily large. Also considered are some of the attempts to model priority in CSP.

Some of the earliest algorithms resolving choice over multiway synchronisation are database transaction protocols such as the two phase commit protocol [9]. Here the choice is between selecting a ‘commit’ event or one or more processes choosing to ‘abort’ an attempt to commit changes to the database. Initially such protocols were blocking. After the commit attempt was initiated, a coordinator would ask the enrolled nodes to commit to the transaction. If all nodes commit in this way then the transaction is confirmed by an acknowledgement otherwise the nodes are informed that they should abort the transaction. In either case the network and the nodes themselves were considered to be reliable and responsive to such requests.

Later incarnations were non-blocking and tolerated faults by introducing the possibility that transactions could timeout [10], these are sometimes referred to as a 3 phase commit protocol. The first phase asks nodes if they are in a position to synchronise (which the coordinator acknowledges), the second involves the processes actually committing to the synchronisation, this being subject to timeouts, the third ensures that the ‘commit’ or ‘abort’ is consistent for all nodes.

The protocols above are limited in that they can be considered to be choosing over two events ‘commit’ and ‘abort’. A more general solution was proposed by McEwan [3] which reduced the choice to state machines connected to a central controller. This was followed by an algorithm which coordinates all multiway synchronisations through a single central *Oracle* [11] and implemented as a library extension for JCSP [5,4] in the form of the *AltingBarrier* class.

All of the above algorithms are incompatible with user defined priority. The database commit protocols are only compatible with priority to the extent that committing to a transaction is favoured over aborting it. The more general algorithms have no mechanism by which priority can be imposed and in the case of JCSP *AltingBarrier*s this incompatibility is made explicit.

There have been many attempts to formalise event priority in CSP. Fidge [12] considers previous approaches which (either statically or dynamically) assign global absolute priority values to specific events, these approaches are considered to be less modular and compositional. Fidge instead proposes an asymmetric choice operator ($\overrightarrow{[]}$) which favours the left operand. Such an operator is distinguished from the regular external choice operator in that

it excludes the traces of the right hand (low priority) operand where both are allowed by the system, i.e. the high priority event is always chosen where possible. While this might be considered ideal, in practice the arbitrary nature of scheduling may allow high priority events to not be ready, even when the system allows it. Therefore low priority events are not excluded in practice in CSP based languages.

However the introduction of readiness tests to CSP by Lowe [13] allow for priority to be modelled as implemented in CSP based languages. Using this model priority conflicts (an inevitable possibility with locally defined relative priority structure) are resolved by arbitrary selection, this is the same result as occurs with JCSP AltableBarriers (albeit with higher performance costs). However, Lowe treats readiness as a binary property of events, in Section 3.1 a case is presented for treating readiness as a (possibly false) assertion that all enrolled processes will be in a position to synchronise on an event in the near future. This distinction allows for processes to pre-emptively wait for multiway synchronisations to occur. Section 2.2 establishes this as being necessary to implement meaningful priority.

2. Limitations of Existing External Choice Algorithms

Existing algorithms offering choice over multiway synchronisation do not offer any mechanism for expressing priority; they offer only *arbitrary* selection (which is all that standard CSP describes). Listed in this section are two (not intuitively obvious) ways in which repeated use of these selection algorithms can be profoundly unfair – although it is worth bearing in mind that CSP choice has no requirement for priority and repeated CSP choice has no requirement for fairness. As such, while these aspects of existing choice resolution algorithms are arguably undesirable, they all constitute valid implementations of external choice.

2.1. Arbitration by Barrier Size

Pre-existing algorithms for resolving choice over multiway synchronisation have in common an incompatibility with priority [4], this means that event selection is considered to be *arbitrary* - in other words no guarantees are made about priority, fairness or avoiding starvation. It is therefore the responsibility of programmers to ensure that this limitation has no adverse effects on their code. While the problems of arbitrary selection may be relatively tractable for code involving channel communications, code containing barrier guards pose extra complications. Consider the following occam-π pseudo-code:

```
PROC P1 (BARRIER a, b)
  ALT
    SYNC a
      SKIP
    SYNC b
      SKIP
:

PROC P2 (BARRIER a)
  SYNC a
:

PROC P3 (BARRIER b)
  SYNC b
:
```

Three different types of processes, one enrolled on ‘a’ and ‘b’, the other two enrolled on either one or the other but not both. Consider a process network containing only P1 and P3 processes:

```
PROC main(VAL INT n, m)
  BARRIER a, b, c:
  PAR
    PAR i = 0 FOR n
      P1 (a, b)
    PAR i = 0 FOR m
      P3 (b)
:
```

In such a network event ‘a’ is favoured. In order for either event to happen all of the processes enrolled on that event must be offering it. Since the set of processes enrolled on ‘a’ is a subset of those enrolled on ‘b’, for ‘b’ to be ready implies that ‘a’ is also ready (although the reverse is not true). It is therefore necessary for all of the P3 processes to offer ‘b’ before all of the P1 processes offer ‘a’ and ‘b’ in order for synchronisation on ‘b’ to be possible (even then the final selection is arbitrary). However as the ratio of P1 to P3 processes increases this necessary (but not sufficient) condition becomes less and less likely.

This state of affairs may however be desirable to a programmer. For example another process in the system may be enrolled on ‘a’ but also waiting for user input. Provided that processes P1 and P3 are looped ad infinitum, ‘a’ may represent a high priority, infrequently triggered event while ‘b’ is less important and is only serviced when ‘a’ is unavailable. A naive programmer may consider that this property will always hold true. However consider what happens if P2 processes are dynamically added to the process network. Initially ‘a’ continues to be prioritised over ‘b’ but once the P2 processes outnumber the P3 processes it becomes more and more likely that ‘b’ will be picked over ‘a’, even if ‘a’ would otherwise be ready.

For this reason a programmer needs to not only be aware of the overall structure of their program in order to reason about which events are selected but also the numbers of processes enrolled on those events. This becomes even more difficult if these numbers of enrolled processes change dynamically.

2.2. Unselectable Barriers

As well as making the selection of events depend (to an extent) on the relative numbers of processes enrolled on competing barriers, existing algorithms for resolving external choice over multiway synchronisation can allow for the selection of certain events to be not only unlikely but (for practical purposes) impossible. Consider the pseudo-code for the following two processes:

```
PROC P1 (BARRIER a, b)
  WHILE TRUE
    ALT
      SYNC a
        SKIP
      SYNC b
        SKIP
:

PROC P2 (BARRIER a, c)
  WHILE TRUE
    ALT
      SYNC a
        SKIP
      SYNC c
        SKIP
:
```

If a process network is constructed exclusively out of P1 and P2 processes then the sets of processes enrolled on ‘a’, ‘b’ and ‘c’ have some interesting properties. The set of processes enrolled on ‘b’ and those enrolled on ‘c’ are both strict sub-sets of those enrolled on ‘a’. Further the intersection of the sets for ‘b’ and ‘c’ is the empty set.

Since choice resolution algorithms (like the Oracle algorithm used in JCSP *AltingBarriers*) always select events as soon as they are ready (i.e. all enrolled processes are in a position to synchronise on the event), this means that for an event to be selected it must become ready either at the same time or before any competing events. However, because ‘a’ is a superset of ‘b’ and ‘c’ it would be necessary for ‘a’, ‘b’ and ‘c’ to become ready at the same time for ‘a’ to be selectable. This is impossible because only one process may make or retract offers at a time and no process offers ‘a’, ‘b’ and ‘c’ simultaneously. It is therefore impossible for ‘a’ to be selected as either ‘b’ or ‘c’ must become ready first.

The impossibility of event ‘a’ being selected in the above scenario holds true for *AltingBarrier* events: each process offers its set of events atomically and the Oracle deals with each offer atomically (i.e. without interruption by other offers). However, this need not happen. If it is possible for processes to have offered event ‘a’ but to have not yet offered event ‘b’ or ‘c’, then ‘a’ may be selected if a sufficient number of processes have offered ‘a’ and are being slow about offering ‘b’ or ‘c’. This gives a clue as to how priority can be introduced into a multiway synchronisation resolution algorithm.

3. Limitations of Existing Priority Models

3.1. Case for Redefining Readiness and Priority

As discussed in Section 2.2, selecting events as soon as all enrolled processes have offered to synchronise can cause serious problems for applying priority to choice over multiway synchronisation. As such meaningful priority may be introduced by allowing processes to pre-emptively wait for synchronisations to occur or by suppressing the readiness of other events in favour of higher priority ones.

Here to *pre-emptively wait* on a given event means to offer only that event and to exclude the possibility of synchronising on any others that would otherwise be available in an external choice. Events which are not part of that external choice may be used to stop a process pre-emptively waiting, for example a timeout elapsing may trigger this. Once a process stops pre-emptively waiting it is once again free to offer any of the events in an external choice. In other words a process waits for the completion of one event over any other in the hope that it will be completed soon, if it is not then the process may consider offering other events.

Waiting in this way requires the resolution of two problems. The first is that if processes wait indefinitely for synchronisations to occur, the network to which the process belongs may deadlock. The corollary to this is that where it is known in advance that an event cannot be selected, it should be possible for processes to bypass waiting for that event altogether (so as to avoid unnecessary delays). The second is that, as a consequence of the first problem, when a process does stop waiting for a high priority event and begins waiting for a lower priority one, it is possible that the higher priority event may become ready again. Here, *ready again* means that the event now merits its set of processes enrolled on it *pre-emptively waiting* for its completion. Thus it must be possible for a process to switch from pre-emptively waiting for a low priority synchronisation to a higher priority one.

While there are almost an infinite number of ways of pre-emptively determining the readiness of any event, it is proposed that *PCOMS* barriers use flags to pre-emptively assert the readiness of enrolled processes. Each process uses its flags (one each per barrier that it is enrolled on) to assert whether or not it is in a position to synchronise on that event in the near future. It is a necessary condition that all enrolled processes assert their readiness for those

processes to begin waiting for that synchronisation to take place. If this condition becomes false during such a synchronisation attempt then that attempt is aborted. Conversely if this condition becomes true then this triggers enrolled processes waiting for lower priority events to switch to the newly available event.

For the purposes of causing synchronisation attempts to be aborted because of a timeout, such timeouts falsify the assertion of the absent processes that they are in a position to synchronise in the near future. Their flags are changed to reflect this, this in turn causes the synchronisation attempt as a whole to be aborted.

In this way high priority events are given every opportunity to be selected over their lower priority counterparts, while the programmer is given every opportunity to avoid wasteful synchronisation attempts where it is known that such a synchronisation is unlikely,

3.2. Case for Nested Priority

While there are positive uses for prioritising some multiway synchronisations over others (graceful termination, pausing, etc.) there may be some circumstances where imposing a priority structure on existing arbitrary external choices can be undesireable. Consider the process network for the TUNA project's one-dimensional blood clotting model [11]. Each SITE process communicates with others through a 'tock' event and an array of 'pass' events, each process being enrolled on a pass event corresponding to itself as well as the two processes in front of it in a linear pipeline. Although the events offered at any given time depend on the SITE process' current state, it is a convenient abstraction to consider that the SITE process offers all events at all times, as in the following pseudo-code:

```
PROC site (VAL INT i)
  WHILE TRUE
    ALT
      ALT n = 0 FOR 3
        SYNC pass[i+n]
          SKIP
      SYNC tock
        SKIP
:
```

Here the SITE process makes an arbitrary selection over the events that it is enrolled on. Now suppose that the SITE processes also offer to synchronise on a 'pause' barrier. This barrier would need to be of higher priority than the other barriers and would presumably only be triggered occasionally by another process waiting for user interaction. A naive way of implementing this could be the following:

```
PROC site (VAL INT i)
  WHILE TRUE
    PRI ALT
      SYNC pause
        SKIP
      PRI ALT n = 0 FOR 3
        SYNC pass[i+n]
          SKIP
      SYNC tock
        SKIP
:
```

Here the SITE process prioritises the 'pause' barrier most highly, followed by the 'pass' barriers in numerical order, followed by the 'tock' barrier. This might not be initially considered a problem as any priority ordering is simply a refinement of an arbitrary selection scheme.

However when more than one process like this is composed in parallel problems begin to emerge, each individual SITE process identified by the 'i' parameter passed to it prefers the 'pass[i]' event over other pass events further down the pipeline. In other words SITE2 prefers 'pass[2]' over 'pass[3]', while SITE3 prefers 'pass[3]' over all others and so on. This constitues a priority conflict as there is no event consistently favoured by all processes enrolled on it.

To paraphrase, each process wishes to select its own 'pass' event and will only consider lower priority events when it is satisfied that its own 'pass' event is not going to complete. Since no processes can agree on which event is to be prioritised there is no event which can be selected which is consistent with every process' priority structure. There are two ways in which this can be resolved. The first is that the system deadlocks. The second is that each process wastes time waiting for its favoured event to complete, comes to the conclusion that the event will not complete and begins offering other events. This second option is effectively an (inefficient) arbitrary selection.

The proposed solution to this problem for *PCOMS* barriers is to allow groups of events in an external choice to have no internal priority but for that group to exist in a wider prioritised context. For the purposes of expressing this as occam-π pseudo-code, a group of guards in an `ALT` block are considered to have no internal priority structure but if that block is embedded in `PRI ALT` block then those events all fit into the wider priority context of the `PRI ALT` block. For example in this code:

```
PROC site (VAL INT i)
  WHILE TRUE
    PRI ALT
      SYNC pause
        SKIP
      ALT
        ALT n = 0 FOR 3
          SYNC pass[i+n]
            SKIP
        SYNC tock
          SKIP
:
```

The 'pause' event is considered to be have higher priority than all other events but the 'pass' and 'tock' events are all considered to have the same priority, thereby eliminating any priority conflict. All processes instead are willing to offer any of the 'pass' or 'tock' events without wasting time waiting for the completion of any one event over any other.

4. Implementation Nomenclature

For the purposes of discussing both the interface and implementation of JCSP *PCOMS* barriers, it is necessary to describe a number of new and extant JCSP classes as well as some of their internal fields or states. A UML class diagram is shown in Figure 1.

PCOMS barrier The generic name for any barrier which is capable of expressing nested priority and which can (when involved in an external choice) optimistically wait for synchronisation to occur (as opposed to requiring absolute readiness).

AltableBarrierBase The name of a specific JCSP class representing a *PCOMS* barrier. An *AltableBarrierBase* contains references to all enrolled processes through their *AltableBarrier* front-ends.

AltableBarrier A JCSP class representing a process's front-end for interacting with an *AltableBarrierBase*. There is exactly one *AltableBarrier* per process per *AltableBarrier-*

Base that it is enrolled on. Henceforth, unless otherwise noted, the term *barrier* is used as a short hand for an *AltableBarrier*. Further an *AltableBarrier*, may in context, refer to the *AltableBarrierBase* to which it belongs. For example a process which selects an *AltableBarrier* also selects the *AltableBarrierBase* to which it belongs.

GuardGroup a collection of one or more AltableBarriers which are considered to be of equal priority.

BarrierFace A class used to store important information about a process' current state regarding synchronisation attempts on AltableBarriers. Includes the *AltableBarrier* (if any) that a process is trying to synchronise on, the local lock which must be claimed in order to wake a waiting process, etc. There is a maximum of one *BarrierFace* per process.

'Status', PREPARED, UNPREPARED and PROBABLY READY Each *AltableBarrier* has a 'status' flag which records whether a process is *PREPARED* or *UNPREPARED* to synchronise on that barrier in the near future. An *AltableBarrierBase* is considered *PROBABLY READY* iff all enrolled processes are *PREPARED*. Being *PROBABLY READY* is a prerequisite for a process to attempt a synchronisation on an *AltableBarrier*.

Alternative An existing JCSP class which is the equivalent of an occam-π `ALT`. Calling its *priSelect()* method causes it to make a prioritised external choice over its collection of Guards.

altmonitor A unique object stored in an *Alternative*. If an *Alternative* (when resolving external choice) checks all of its Guards and finds none of them are ready then the invoking process calls the *wait()* method on the *altmonitor*. The process then waits for any of the Guards to become ready before being woken up by a corresponding *notify()* call on the *altmonitor*.

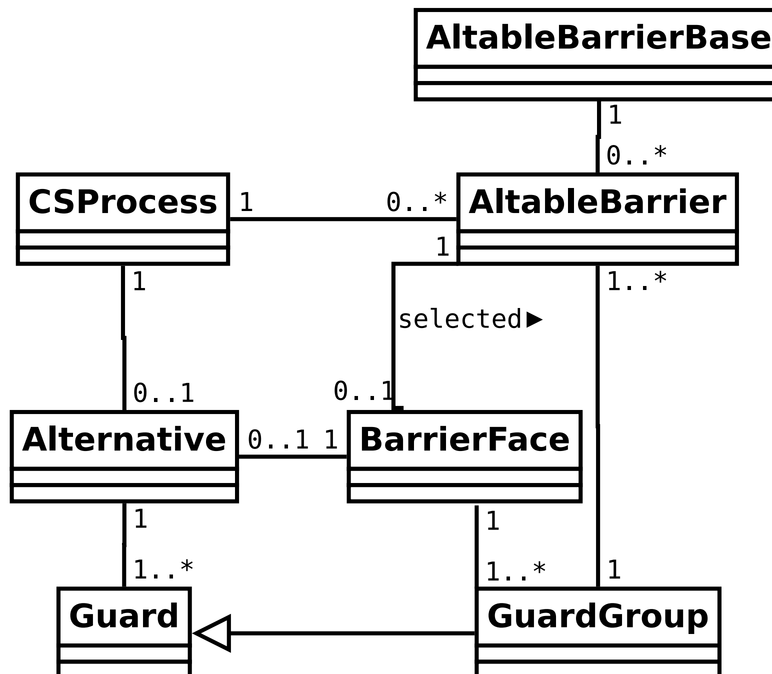

Figure 1. UML diagram showing how the relationship between new and existing JCSP classes.

5. Description of PCOMS Interface

This section illustrates the interface programmers use to interact with AltableBarriers. The source code for all of the classes described in this section can be downloaded from a branch in the main JCSP Subversion repository [14]. All of these classes are contained in the `org.jcsp.lang` package.

5.1. Compatibility with Existing JCSP Implementation

The *AltableBarrier* class, although not directly extending the Guard class, is nevertheless designed to be used in conjunction with the *Alternative* class in JCSP. A single object shared between all enrolled processes of the class *AltableBarrierBase* is used to represent the actual *PCOMS* barrier. Each process then constructs its own *AltableBarrier* object, passing the *AltableBarrierBase* object to the constructor. This creates an individual front-end to the barrier for that process and enrols that process on the barrier.

The *AltableBarrier* is included as a Guard in an *Alternative* by passing an array of AltableBarriers to the constructor of a *GuardGroup* object. This class extends the Guard class and functions as a collection of one or more AltableBarriers.

```
\\construct a new barrier
AltableBarrierBase base = new AltableBarrierBase();
\\ enrol a process on a barrier
AltableBarrier bar = new AltableBarrier(base);
\\ create a GuardGroup containing only one barrier
GuardGroup group = new GuardGroup(new AltableBarrier[]{bar});
```

5.2. Mechanism for Expressing Nested Priority

Guards are passed to an *Alternative* constructor as an array, the order in which the elements are arranged determines the priority ordering. Since the *GuardGroup* class extends Guard, the relative priority of AltableBarriers is determined by the position of the *GuardGroup* to which they belong.

However a single *GuardGroup* can contain more than one *AltableBarrier*, such barriers have no priority ordering within the *GuardGroup* (the selection process is detailed later but may be considered arbitrary). In this way a group of barrier guards with no internal priority between themselves can be nested within a larger priority structure

```
\\ various AltableBarriers intended to have different priorities.
\\ assume these variables have real AltableBarrier
\\ objects assigned
AltableBarrier highBar, midBar1, midBar2, lowBar;
\\ create 3 different GuardGroups, one for each priority level.
\\ note that mid has two AltableBarriers which are of
\\ equal priority
GuardGroup high = new GuardGroup(
        new AltableBarrier[]{highBar}
);
GuardGroup mid = new GuardGroup(
        new AltableBarrier[]{midBar1, midBar2}
);
GuardGroup low = new GuardGroup(
        new AltableBarrier[]{lowBar}
);
Guard[] guards = new Guard[]{high, mid, low};
Alternative alt = new Alternative(guards);
```

5.3. Mechanisms for Manipulating Readiness

As explained earlier (Section 3.1), the ability to express meaningful priority over multiway synchronisation requires the ability to express a future ability to engage on an event as well as the ability to correct for false positive and negative readiness tests.

With regard to the former, a *PCOMS* barrier is considered ready if all of the enrolled processes have advertised the fact that they are able to synchronise on that barrier in the near future. To this end the all *AltableBarrier* objects have a flag indicating whether a process is *PREPARED* or *UNPREPARED* to synchronise on that barrier. For a synchronisation to be attempted all enrolled process must be *PREPARED*. These flags do not reflect whether or not a process is actually offering an event at any given moment. Instead it indicates whether or not (in the programmer's opinion) that process will be in a position to offer that event within a reasonable time frame and that the process network as a whole will not deadlock if other processes act on this information.

While a process is evaluating an *Alternative*'s *priSelect()* method, the state of this flag is managed automatically. A process becomes *PREPARED* to synchronise on a given barrier as soon as it is encountered in that *Alternative*, likewise it is automatically made *UNPREPARED* if a synchronisation attempt is made but that process fails to engage on that event before a timeout elapses (this state persisting until that process actually is in a position to engage on that event again). At all other times a user defined default state holds for each individual *AltableBarrier* object. It is however possible for the programmer to override this state temporarily (i.e. until the state is changed automatically) by calling the *AltableBarrier*'s *setStatus()* method or more permanently by overriding its default state by calling its *setDefaultStatus()* method.

In general any process which regularly evaluates an *Alternative* containing a given *AltableBarrier* such as server processes should set this default to *PREPARED*. Conversely processes which act as clients or which wait for user or network input (and thus may be significantly delayed before attempting a synchronisation with a barrier) should set this default to *UNPREPARED*. While changes to the default after construction are left at the programmer's discretion, such changes should be unnecessary unless a significant change in the behaviour of a process occurs.

```
AltableBarrier bar1 =
        new AltableBarrier(base, AltableBarrier.UNPREPARED);
AltableBarrier bar2 =
        new AltableBarrier(base, AltableBarrier.PREPARED);

bar1.setStatus(AltableBarrier.PREPARED);
bar2.setDefaultStatus(AltableBarrier.UNPREPARED);
```

5.4. Discovery and Acknowledgement of Events After Selection

Once an *AltableBarrier* has been selected by a call to the *priSelect()* method, the index returned by that method will indicate the *GuardGroup* object to which that barrier belongs. Calling the *lastSynchronised()* method on that *GuardGroup* will reveal the specific *AltableBarrier* selected. By this point the actual synchronisation on the barrier will have taken place. Therefore, unlike with JCSP channel synchronisations, it is unnecessary for the programmer to do anything else to complete or acknowledge the synchronisation having occurred.

To paraphrase, an *AltableBarrier* is used in the same way as an *AltingBarrier* with two exceptions. The first being that AltableBarriers need to be enclosed in a *GuardGroup* to which it belongs. This *GuardGroup* must be interrogated if the selected barrier is ambiguous. The second is that priority cannot be expressed using *AltingBarrier*s.

```
int index = alt.priSelect();

Guard selectedGuard = guards[index];
AltableBarrier selected = null;
if (selectedGuard instanceof GuardGroup) {
        GuardGroup group = (GuardGroup) selectedGuard;
        selected = group.lastSynchronised();
}

\\ The synchronisation has already taken place at this point,
\\ no further action is required to acknowledge the event.
```

5.5. Current Limitations

JCSP AltableBarriers (via an enclosing *GuardGroup* object) can be used with any number of existing JCSP Guards in any combination with two restrictions. The first is that no *Alternative* object may enclose both a *GuardGroup* and a *AltingBarrier* (the latter being the name of a class which implements the old Oracle algorithm). Code required to ensure consistency of selection for *AltingBarrier*s can cause inconsistency for the new AltableBarriers. The second restriction is that only the *priSelect()* method of the *Alternative* class is considered safe for use with AltableBarriers, behaviour when using the *select()* or *fairSelect()* methods is not considered here.

It should also be noted that the existing *AltableBarrier* implementation lacks any mechanism for allowing processes to resign from a barrier. This restriction is not intended to be permanent. In the interim processes wishing to resign from an *AltableBarrier* should spawn a new process and pass it the unwanted *AltableBarrier* object, this process should loop infinitely, offering to synchronise on that barrier with each iteration.

Finally, AltableBarriers are incompatible with the use of any boolean preconditions.

6. Description of PCOMS Algorithm

This section details the inner workings of the *PCOMS* algorithm as applied to JCSP. The algorithm is inspired in part by the 3 phase commit protocol [10]. Specifically the algorithm can be broken down into 3 distinct phases. The first concerns establishing whether or not an *AltableBarrier* (or group of AltableBarriers) is in a position for enrolled processes to begin pre-emptively waiting for a synchronisation to occur and selecting such a barrier in a manner consistent with priority ordering. The second phase involves waiting for the synchronisation itself and includes details of mechanisms for ensuring consistency of selection between processes as well as of the mechanisms for aborting synchronisation attempts. The third phase involves ensuring that any synchronisations are consistently reported by all processes. Details are also given for processes which have begun waiting on the '*altmonitor*', an object unique to each instance of the *Alternative* class used to wake processes waiting for any (even non barrier) guards to become ready.

While the possibility that this algorithm could be simplified should not be ruled out, the relative complexity of this algorithm serves to prevent deadlock. Much of the complexity is required for compatibility with existing *Alternative* Guards. For example, special provisions must be made for processes waiting on the *altmonitor* object. This is because such processes may be woken up by either a successful barrier synchronisation or by a conventional JCSP Guard.

6.1. Phase 1: Readiness Testing

Figure 2 outlines the logic. All Guards in a JCSP *Alternative* have their readiness tested by a call to their *enable()* method: calling *enable()* on a *GuardGroup* initiates readiness tests on all of the AltableBarriers that that *GuardGroup* contains. A call to the *enable()* method of a *GuardGroup* returns true iff an attempt to synchronise on an *AltableBarrier* has been successful.

When enable is called on a *GuardGroup*, it claims a global lock: this lock is required for all reading and writing operations to all data related to *AltableBarrier* synchronisations. This lock is not released until either a process begins waiting for a synchronisation to occur or the invoking *enable()* method has been completed and is ready to return.

Once the global lock has been claimed, the process sets the status flag of all of the AltableBarriers contained in the *GuardGroup* (and all of those contained in higher priority GuardGroups[1]) to *PREPARED*.

The next step is to select a barrier on which to attempt synchronisation. For each *GuardGroup* encountered in the Alternative so far, in priority order, all of the AltableBarriers in each *GuardGroup* are examined to see if they are *PROBABLY READY*. If no AltableBarriers are *PROBABLY READY* then the next *GuardGroup* is examined. If no AltableBarriers are ready in all of the GuardGroups under consideration, then the *enable()* method releases the global lock and returns false.

If one or more AltableBarriers are found to be *PROBABLY READY*, then they are each tested to see if any have been selected by other processes. If some of them have, then those that have not are eliminated from consideration for now. In either case, an *AltableBarrier* is arbitrarily selected from the list of *PROBABLY READY* barriers which remain. In this way, an *AltableBarrier* is selected which is *PROBABLY READY*, of equal or greater priority to other possible barriers and is, if possible, the same choice of barrier as selected by the process' peers.

6.2. Phase 2: Awaiting Completion

The process holding the global lock now has an *AltableBarrier* on which it intends to attempt a synchronisation. It is already the case that this barrier is one of (or the) highest priority barriers currently available and that (where applicable) it is also a barrier which has been selected by other processes. However, there may be other processes enrolled on this barrier currently attempting to synchronise on other lower priority barriers. In order for the barrier synchronisation to complete, it is necessary for those processes waiting on other barriers to be stolen (see Section 6.2.1). These processes, where they could be stolen, continue to wait but are now waiting for the 'stealing' barrier to complete. See Figure 3.

Having ensured maximum consistency between all processes attempting barrier synchronisations, the process holding the global lock checks to see if it is the last process required to complete the barrier synchronisation. If it is, then the waiting processes are informed of the successful synchronisation and woken up (see Section 6.3). If not, then the process will need to begin waiting – either for a successful synchronisation attempt or for the synchronisation to be aborted.

If this is the only process currently attempting to synchronise on the barrier, then a timeout process is started (see Section 6.2.2) to ensure that any synchronisation attempt is not continued indefinitely. The *BarrierFace* is then updated to reflect the currently selected

---

[1]During the time between the evaluation of one *GuardGroup* and another it is possible for a synchronisation attempt on an *AltableBarrier* to have timed-out. In such a case the currently running process may have had its status flag (associated with that barrier) set to *UNPREPARED*. Given that this process is now once again in a position to offer that event, it is necessary for such flags to be reset to *PREPARED*.

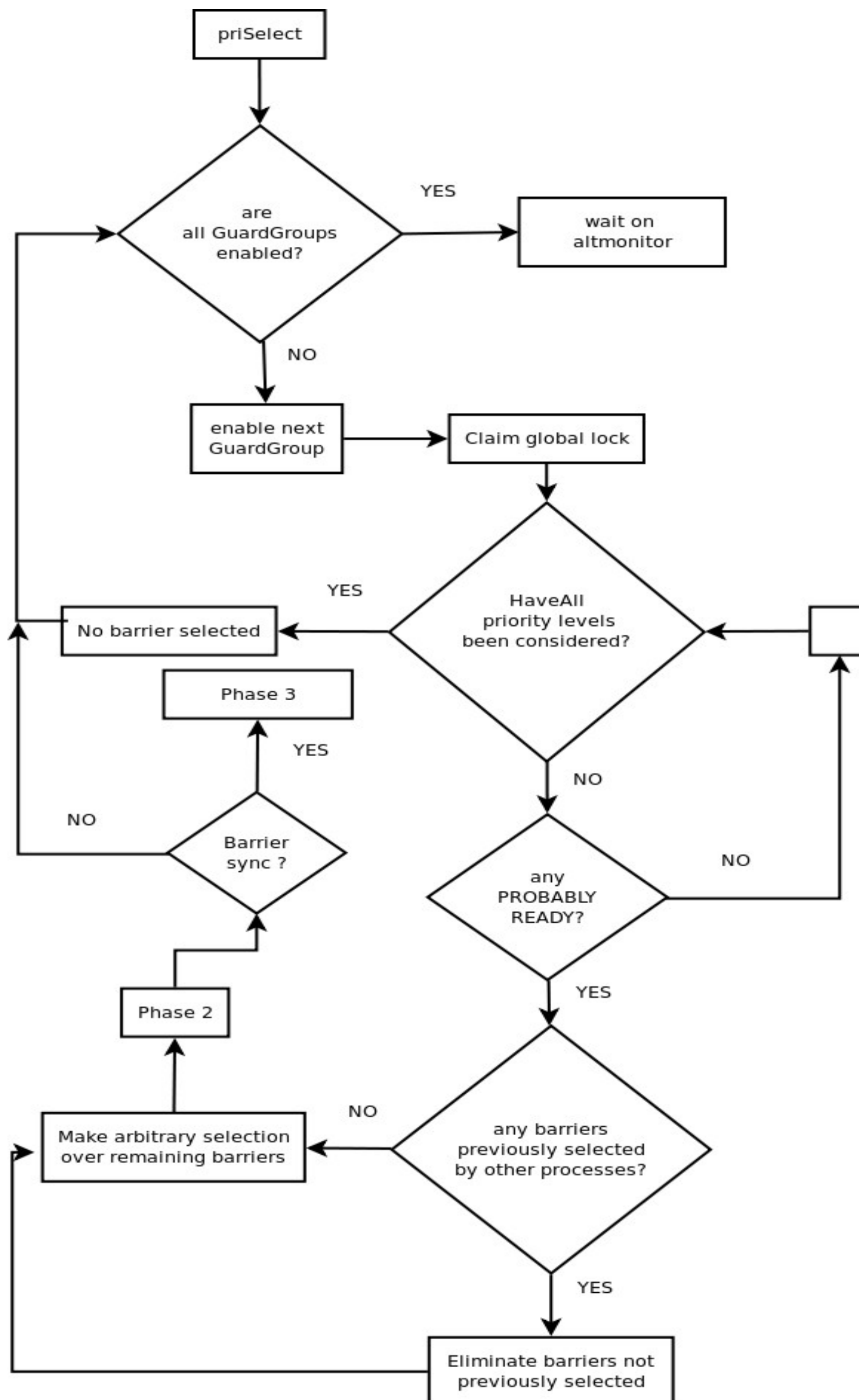

Figure 2. Flow Chart showing Phase 1.

barrier and an object representing a local lock used to wake the process once it has begun waiting. The object used for this local lock is the enclosing *Alternative* object itself: this has the virtue of being unique to each process and of being distinct from the *Alternative*'s *altmonitor* (this is to avoid the process being woken up by non-barrier guards becoming ready).

Then, the process claims its local lock and, afterwards, releases the global lock. It then calls the local lock object's *wait()* method, meaning that the process will sleep either until a barrier synchronisation is successful or its synchronisation attempt is aborted. During

this waiting time a process may be stolen (see Section 6.2.1) any number of times. For the purposes of ensuring deadlock freedom, it is important to note that all processes which wait for synchronisations to complete – as well as processes which wake them up – have *always* claimed the global lock first, then claim the local lock before releasing the global lock. When waiting processes are woken up, they initially own their local lock and then claim the global lock; this inversion in the order in which locks are claimed can potentially cause deadlock. To counter this, there are two strict conditions imposed on waking processes:

1. A process must first be waiting before another process can attempt to wake it.
2. The *BarrierFace* of each process has a 'waking' flag which is set to true once a process has woken it up. No process will attempt to wake a process with a true 'waking' flag.

This means that locks are always claimed in the order global-then-local until a process is woken up, after which locks are claimed in the order local-then-global.

In summary a process attempting a synchronisation will do one of two things. If it is the last process required to complete a synchronisation, it will do so. Otherwise it will begin waiting for the synchronisation to complete or for the attempt to be aborted. In any case, after this phase has been completed, the process in question will know whether or not it successfully synchronised on a barrier and, if so, which one. If synchronisation was successful, then phase 3 (Section 6.3) ensures that this is consistently reported by all processes involved.

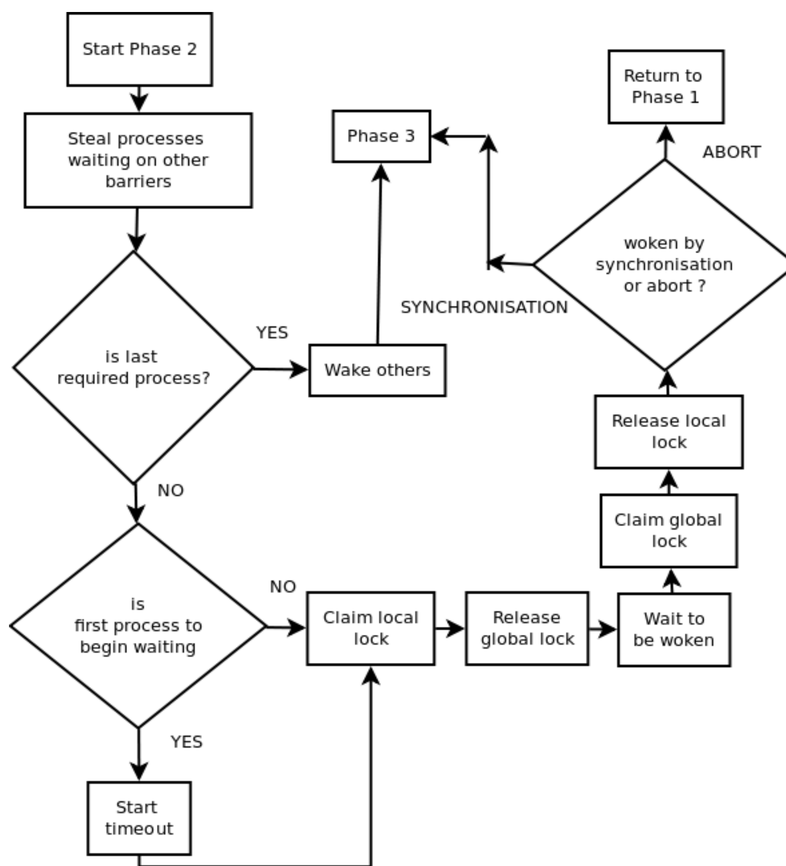

Figure 3. Flow Chart showing Phase 2.

6.2.1. Stealing

Stealing is the way in which processes enrolled on a given barrier but currently waiting for the completion of different barriers are switched from waiting on the latter to the former. For each of the processes enrolled on the stealing barrier the following tests are run, failing any means that the process isn't stolen:

1. Is the process currently evaluating an *Alternative*?
2. Is it currently waiting for the completion of another barrier?
3. Does the stealing barrier have and equal or higher priority than the old barrier from the point of view of the process being stolen? [2]
4. Is the process' 'waking' flag still false?

If these conditions are met then the process is stolen by simply changing the *AltableBarrier* object recorded in the process' *BarrierFace*.

6.2.2. Timeouts

A timeout process is created when the first process enrolled on a barrier begins waiting for its completion, its purpose is to abort synchronisation attempts on a barrier which take too long.

When created and started in its own Thread a timeout process waits for a time period dependant on the number of processes enrolled on its corresponding barrier. Currently this time period is 500 milliseconds multiplied by the number of enrolled processes. This formula is entirely arbitrary but has proved to be a generous estimation of the time required to complete barrier synchronisations of any size. A more detailed analysis of *PCOMS* barrier performance would be required to minimise the time spent waiting for false-positive synchronisation attempts.

When the timeout period has elapsed the timeout process claims the global lock and examines an internal flag, the state of which depends on whether or not the barrier synchronisation was successful while the timeout process was asleep, if it was then it releases the global lock and terminates. If the synchronisation attempt has yet to complete then the timeout process aborts the synchronisation attempt in the following way.

Each process enrolled on that barrier but which is not currently attempting to synchronise on it has its status flag (associated with the timed-out barrier) set to *UNPREPARED*. In other words, that process' assertion that it will synchronise on that barrier in the near future has been proven false therefore it is amended to *UNPREPARED* until such time as that process is in a position to synchronise on the barrier.

Changing the status of some processes to *UNPREPARED* means that the barrier as a whole is no longer *PROBABLY READY*, such a change is the only way in which synchronisation attempts are aborted. All processes currently waiting on the aborted barrier have their *BarrierFace* objects amended to reflect that they are no longer waiting for any barrier. Normally, these processes also have their 'waking' flags set to true and are then awoken. If any of these processes are waiting on the *altmonitor* (see Section 6.4), they are not awoken.

Currently there is no mechanism for the programmer to set or terminate these timeouts manually nor to change the amount of time that an event takes to timeout.

6.3. Phase 3: Ensuring Consistency

Having progressed past phases 1 and 2, a process will have selected a barrier to attempt a synchronisation on and will have either succeeded or failed to synchronise (in the interim it may

[2] A process waiting for the completion of a barrier may be stolen by a barrier which the process considers to be of equal priority. This is allowed because the process which initiated the stealing may have a specific priority ordering (where the process being stolen does not) or the stealing process may not be enrolled on the same set of events.

have been stolen by another barrier). In either case, the result must be acted on such that the process either continues to searching for another guard to select or acknowledges a successful synchronisation and ensures that the acknowledgement is consistent for all processes.

At this stage, a process will have access to its *BarrierFace* which will either contain the *AltableBarrier* on which the process has synchronised or will contain a null value in its place. If the latter is the case, then the synchronisation attempt was aborted, the process moves back to phase 1 and either attempts a new synchronisation or (if no AltableBarriers are *PROBABLY READY*) the *enable()* method terminates returning false.

If the process did synchronise, a number of things need to be done to ensure that this is reported correctly. The most important complication is that the *enable()* method invoked belongs to a specific *GuardGroup*, which in turn represents one or more *AltableBarrier* objects contained within. However, because a process may be stolen by a barrier in another *GuardGroup*, the Guard that the *Alternative* selects may be different from the *GuardGroup* whose *enable()* method has been called.

The selected *AltableBarrier* has a reference to the *GuardGroup* which contains it: this *GuardGroup* has a field called 'lastSynchronised' and the selected *AltableBarrier* is assigned to this field. Whether or not the currently executing *GuardGroup* contains the selected *AltableBarrier*, the global lock is released and the enable method returns true. Returning true here causes the previously enabled Guards to be disabled in reverse order.

The *disable()* method of a *GuardGroup* (which also begins by claiming the global lock) changes the status of all the AltableBarriers it contains from *PREPARED* back to its default value. If the *GuardGroup*'s 'lastSynchronised' field has been set to a non-null value (i.e. the selected *AltableBarrier* belongs to this *GuardGroup*), then the executing process releases the global lock and synchronises on a 'gatekeeper' barrier (this being a Barrier object with the same set of enrolled processes as its corresponding *AltableBarrier*). This prevents synchronised processes from proceeding until they have all been woken up and have executed the important parts of their *disable()* methods. The *disable()* method returns true iff its 'lastSynchronised' field is set to a non null value.

The *Alternative* class has also been subtly altered such that if a *GuardGroup*'s *disable()* method ever returns true, then that *GuardGroup*'s index is returned by the *priSelect()* method in preference to any non-barrier Guards which may have become ready in the interim.

In this way, all processes that have successfully synchronised on a barrier will have stopped their *Alternative*'s enable sequence and begun disabling all previously enabled Guards. Only the *GuardGroup* that contains the successful *AltableBarrier* will return true when its *disable()* method is called; no process will be able to proceed until all other processes enrolled on that barrier have also woken up (this prevents processes from waking up, acknowledging the successful synchronisation and then immediately selecting the same event again in the same *Alternative*). The *Alternative* class itself has been subtly altered to prevent the readiness of non-barrier Guards from taking precedence over a *GuardGroup*.

6.4. Behaviour when Waiting on the Altmonitor

When a process begins waiting for a barrier synchronisation during a call to *enable()* in a *GuardGroup*, that process can only be woken by the success or failure of barrier synchronisations. However, when an *Alternative* has enabled all of its Guards, it begins waiting on its *altmonitor*; when any of the enabled Guards become ready, the process is woken up (this includes non-barrier Guards). As such, the way in which processes waiting on their *altmonitor* are dealt with is subtly different. The following is a list of those differences:

1. After the last *GuardGroup* in the *Alternative* has its *enable()* method called, the global lock is *not* released. It remains claimed until just before the process begins waiting on the *altmonitor*. This eliminates the possibility that another process may attempt to

steal it in the interim, since this process is no longer in a position to initiate synchronisation attempts it must always be in a position where it can be stolen.
2. Prior to waiting on the *altmonitor*, the process' *BarrierFace* indicates that the *altmonitor* is its local lock (for the purposes of waking the process) and that it is currently not attempting to synchronise on any event.
3. While the process is waiting, it is not possible for aborted synchronisation attempts to wake it. Only a successful synchronisation attempt or a non-barrier Guard becoming ready will wake the process.
4. When waking up, the process must immediately claim the global lock in order to check whether or not a barrier synchronisation has occurred. If it has, then the process's *BarrierFace* sets its 'waking' flag to true. If it has not, then the possibility remains open for any given barrier to be selected until such time as its *GuardGroup*'s *disable()* method is called.

These changes modify the behaviour associated with waiting for barrier synchronisation to allow for possibility of non-barrier guards being selected, while eliminating risks of inconsistency and / or deadlock.

7. Testing

This section outlines some of the tests run to build confidence in the deadlock freedom of *AltableBarriers*, as well as to compare *AltableBarriers* with the previous *AltingBarrier*'s algorithm. The source code for all of these tests is available in a branch of the main JCSP subversion repository [15]. All of these tests are part of the `org.jcsp.demos.altableBarriers` package. For brevity the pertinent sections of all of the test programs are rendered as occam-π pseudo-code.

For the purposes of assessing performance, it should be noted that at the time of writing, the source code for several *AltableBarrier* related classes contain a significant quantity of debugging statements sent to the standard output. Also no attempts have yet been made to optimise any of the source code.

7.1. Stress Testing

Since a formal proof of deadlock has not been attempted, the VisualDemo class exists as a means of stress testing as much of the *AltableBarrier*'s functionality as possible. It is designed to test the responsiveness of processes to infrequently triggered high priority events, compatibility with existing channel input guards as well as the ability to permit the arbitrary selection of nested low priority events. The process network (Figure 4) centres around processes of following type, connected in a ring via *AltableBarrier*s labelled 'left' and 'right':

```
PROC node (BARRIER pause, left, right, CHAN SIGNAL mid)
  WHILE TRUE
    PRI ALT
      SYNC pause
        SYNC pause
      mid ? SIGNAL
        SKIP
      ALT
        SYNC left
          SKIP
        SYNC right
          SKIP
:
```

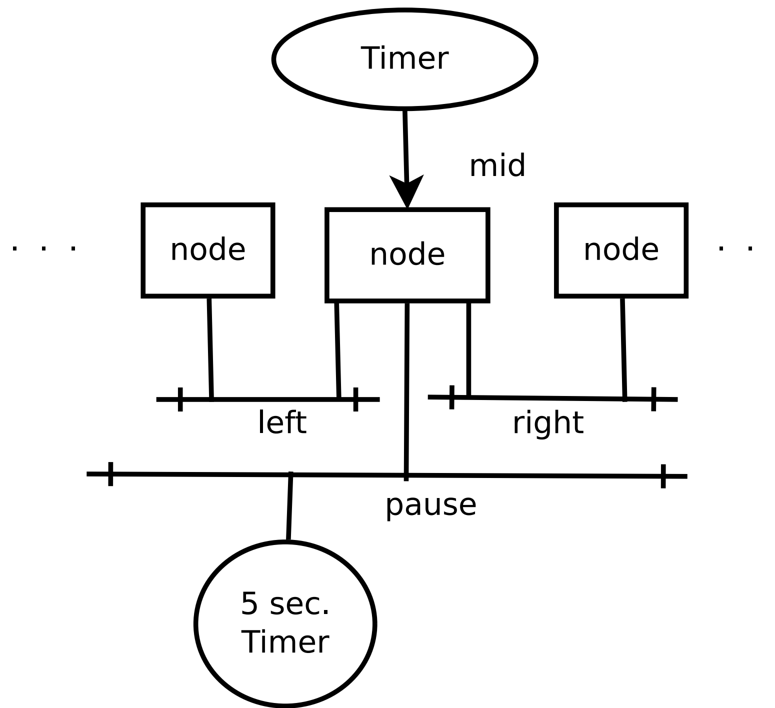

Figure 4. Process diagram showing the way in which node processes are connected in the VisualDemo class

As well as all 'node' processes, the 'pause' barrier is enrolled on by another process which defaults as *UNPREPARED*. In an infinite loop it waits for 5 seconds before offering only the 'pause' barrier. As such, every 5 seconds all node processes synchronise on the 'pause' barrier and then wait a further 5 seconds before being unpaused.

Each process is also connected via its 'mid' channel to another process (one each per node). This process, in an infinite loop, waits for a random timeout between 0 and 10 seconds then sends a signal to its corresponding node process via its 'mid' channel. Thus, when not synchronising on the 'pause' barrier, a node process may synchronise on an ordinary channel communication at random but relatively regular intervals.

When not synchronising on the 'pause' or 'mid' events, node processes synchronise with either of their neighbouring node processes. The selection is arbitrary however where one of the node's neighbours has synchronised on its mid channel, the node process selects the other neighbour (no excessive waiting for the other process to synchronise on its event).

As the name suggests, the VisualDemo class offers a GUI interface which shows these events happening graphically in real time. Although no timing analysis of this system has been attempted, at high numbers of processes (˜100) the 'pause' barrier can take a palpably long time to complete a synchronisation (>1 second after the event first becomes available).

This test can be left to run over several days and has yet to deadlock. While this does not eliminate the possibility of deadlock it is the most complete test of the capabilities of AltableBarriers devised at the time of writing. As such the algorithm is considered to be provisionally deadlock free.

7.2. Comparison with Oracle

To compare the relative performance of *AltingBarrier*s with *AltableBarriers*, a process network consisting of node processes connected in a ring is constructed. The number of node processes in the network is determined by a 'PROCESSES' field. Each node process is connected to a number of processes ahead of it by barriers, the number of nodes it is connected to is determined by the 'OVERLAP' field. Therefore each node is enrolled on 'OVERLAP'

number of barriers, this connects itself with ('OVERLAP'-1) processes ahead of it in the process ring. The pseudo-code for each node is as follows:

```
PROC node (VAL INT id, []BARRIER bars)
  INITIAL INT count IS 0:
  INT start.time, end.time:
  TIMER tim:
  SEQ
    tim ? start.time
    WHILE TRUE
      SEQ
        ALT i = 0 FOR OVERLAP
          SYNC bars[i]
            count := count + 1
        IF
          ((count > ITERATIONS) AND (id = 0))
            SEQ
              tim ? end.time
              out.write ((end.time - start.time), out!)
              KILL  -- terminate all procs, test proc has finished
          TRUE
            SKIP

:
```

When the process network is started each node makes an arbitrary selection over the barriers that it is enrolled on. It then increments a counter for every iteration of this choice. Once the first node in the network (the node with an ID of 0) has reached a fixed number of iterations the entire network is terminated. The amount of time elapsed between the process network starting and it terminating can be used to compare the performance when the barriers are implemented as *AltingBarrier*s versus when they are implemented as AltableBarriers.

The first set of results has a fixed ring of 50 processes, completing 100 iterations. The number of processes to which each node was connected was varied to highlight its effect on speed at which these external choices are resolved. A network using *AltingBarrier*s is tested as is one using AltableBarriers where all processes default to *PREPARED*, finally a network using AltableBarriers where all processes default to *UNPREPARED* is also used.

Table 1. Time (ms) for 50 processes to complete 100 iterations

Overlap	*AltingBarrier*	*PREPARED*	*UNPREPARED*
2	250	12867	19939
3	294	13622	33652
4	303	14093	57939

It is immediately apparent that the existing JCSP *AltingBarrier* algorithm is approximately two orders of magnitude faster than both versions of the *AltableBarrier* algorithm. The degree to which this difference is due to inherent algorithm complexity versus debugging statements, spawning of extra processes and a lack of optimisation is unclear. A detailed analysis of the effects of these factors is beyond the scope of this paper. Both the *AltingBarrier* and '*PREPARED*' networks show modest increases in their completion times as the set of barriers evaluated increases whereas the '*UNPREPARED*' network shows a more dramatic increase. This discrepancy may be due to the need of the '*UNPREPARED*' nodes to examine (and initially reject as unready) all barriers that it encounters until all enrolled processes are in a position to synchronise. Conversely the '*PREPARED*' nodes will select a barrier to attempt a synchronisation with immediately.

The next experiment uses the same *AltingBarrier*, '*PREPARED*' and '*UNPREPARED*' set up as the previous one. However the number of barriers each node is enrolled on is limited to two, the number of processes in the ring is instead varied to examine its effect on performance. As before, 100 iterations are required to terminate the process network. Here,

Table 2. Time (ms) to complete 100 iteration for processes overlapped by two

Num processes	*AltingBarrier*	*PREPARED*	*UNPREPARED*
25	70	5818	13218
50	111	11066	28545
75	330	17957	34516
100	638	24432	44308

the *AltingBarrier* network shows a steeper (possibly n*n) relationship between the number of processes and completion time. The two *AltableBarrier* implementations show a steadier (possibly linear) relation to the number of processes. As before the '*PREPARED*' network outperforms the '*UNPREPARED*' one.

In both experiments the older *AltingBarrier* algorithm is significantly faster than networks using AltableBarriers. In both experiments nodes which defaulted to being '*PREPARED*' to synchronise on their barriers outperformed those which were '*UNPREPARED*'.

7.3. Priority Conflict Resolution

The pre-existing JCSP *AltingBarrier* class lacked any mechanism for expressing priority over events. By adding such mechanisms the *AltableBarrier* class makes it possible for the unwary programmer to introduce priority conflicts. Since the priority of events in an external choice are determined locally, it is possible that these priorities can be defined in such a way as to conflict with eachother. To test the behaviour of AltableBarriers under these conditions and to ensure that such code results in an arbitrary choice, the ConflictTest class creates a network of processes like the following:

```
PROC P1 (BARRIER a, b)
  WHILE TRUE
    PRI ALT
      SYNC a
        SKIP
      SYNC b
        SKIP
:

PROC P2 (BARRIER a, b)
  WHILE TRUE
    PRI ALT
      SYNC b
        SKIP
      SYNC a
        SKIP
:
```

Both P1 and P2 are enrolled on barriers 'a' and 'b'. P1 processes prefer to synchronise on 'a' over 'b', while the opposite is true of P2. In both cases all processes are considered to be *PREPARED* to synchronise on both barriers. So long as the process network as a whole contains at least one P1 and P2 processes the behaviour of the program is the same.

All P1 processes immediately begin to wait pre-emptively for event 'a' to complete while all P2 processes wait for 'b'. Both sets of processes deadlock until one of the barrier

synchronisation attempts times out, as an example we will presume that ‘a’ times out first. As such all processes not waiting for it to complete (all P2 processes) have their status with regard to ‘a’ set to *UNPREPARED*, all P1 processes then abort their synchronisation attempt on ‘a’. Since all P1 processes have abandoned waiting for ‘a’, they are now in a position to consider ‘b’. Either all P1 processes will synchronise on ‘b’ before its synchronisation attempt times out or ‘b’ will timeout and there will be a brief interval during which both ‘a’ and ‘b’ will be considered to be not *PROBABLY READY*. During this period a number of processes will reassert their readiness to synchronise on both barriers and begin waiting on the ‘altmontior’ until either event is ready. Since it will always be the case that there will be at least one process not waiting on the *altmonitor*, there will always be at least one process capable of arbitrating the conflict.

Any further iterations of this choice are likely to be resolved in the same way without the initial delay. Again assuming that ‘b’ was selected over ‘a’, all P2 processes are still considered *UNPREPARED* to synchronise on ‘a’, since they have have not encountered a guard containing ‘a’ they have no opportunity to reset their status flag to their default of *PREPARED*. This means that all P2 processes begin waiting on ‘b’ as usual. All P1 processes, seeing that ‘a’ is not *PROBABLY READY*, skip ‘a’ and immediately synchronise on ‘b’.

This means that although priority conflicts can be (and are) resolved as arbitrary selections, there can be significant performance delays associated with making that choice. It is therefore recommended that nested priority be used to avoid delays caused by priority conflicts. If both P1 and P2 consider ‘a’ and ‘b’ to be of the same priority then there are no delays in making a selection.

8. Discussion

Given the testing (Section 7) performed so far it is possible to provisionally conclude that process networks using AltableBarriers are robust and that they are not vulnerable to priority conflicts. The comparison tests with the existing *AltingBarrier* algorithm reveals that AltableBarriers should be avoided for performance reasons where the ability to prioritise barrier events is not required.

If priority is required and if performance is not an issue, AltableBarriers are useful and offer trivial or manageable delays for modest process networks.

8.1. Future Work

Existing tests have already established that AltableBarriers can be used to atomically pause or terminate process networks and that (using nested priority) this need not affect the existing priority framework or introduce priority conflict. This section details as yet untested patterns for ensuring fairness, avoiding starvation and (possibly) affecting the probability of events being selected.

8.1.1. Fair Alting

Where nested priority is used, selection of the barriers within that block is considered to be arbitrary, therefore no guarantees are made about the fairness of that selection in general. Similarly fair alting cannot be achieved in the same way that is achieved using channel guards (imposing a priority ordering on all events with the last selected guard last). This is because imposing a priority ordering on all barriers where those barriers have overlapping sets of enrolled processes leads to priority conflicts.

To get around this problem, code of the following type could be used to ensure a degree of fairness:

```
PROC fair.alter ([]BARRIER bars)
  BARRIER last.selected:
  WHILE TRUE
    PRI ALT
      ALT i = 0 FOR SIZE bars
        (NOT (bars[i] = last.selected)) && SYNC bars[i]
          last.selected := bars[i]
      SYNC last.selected
        SKIP
:
```

Care must be taken to chose an initially consistent 'last.selected' for all processes, it is also important to note that preconditions are not currently compatible with AltableBarriers and that the 'last.selected' barrier would need to be fully removed from the nested priority block.

However this system ensures that all processes consider the last selected barrier event to be of a lower priority than its peers without imposing a conflict prone priority structure on the rest of the barriers. Further because the selection of the low priority barrier is done on the basis of the last selected barrier, this change in the priority ordering is guaranteed to be consistent for all processes enrolled on that barrier, therefore there this does not cause a priority conflict.

While this may prevent any one event dominating all others, it may not however guarantee complete fairness. The possibility exists that in sets of overlapping events larger than two, two events may consistently alternate as the last selected barrier.

8.1.2. Partial Priority

As well allowing for a general priority structure while avoiding priority conflicts, nested priority may be useful in affecting the probability of one or more events being selected. This proposed scheme will be known as *partial priority* from this point onwards. Consider the simplified model of the SITE processes in the TUNA blood clotting model [11] in Section 3.2, no priority ordering is imposed on any of the events. In the case of the old JCSP *AltingBarrier*s this meant that the 'pass' events were always selected over the 'tock' event. Using AltableBarriers also allows for arbitrary selection of events, in practice (and in the absence of preferences by other processes) the event initially selected by any process is the first one listed in a *GuardGroup*.

As such if the 'pass' events are occur before the 'tock' event in a *GuardGroup*, the 'pass' events are naturally favoured over the 'tock' event. Now consider what happens if one process, selected at random, prioritises 'tock' over the 'pass' barriers:

```
PROC site ([]BARRIER pass, BARRIER tock)
  WHILE TRUE
    PRI ALT
      SYNC tock
        SKIP
      ALT i = 0 FOR SIZE pass
        SYNC pass[i]
          SKIP
:
```

Since the behaviour of processes with regards to priority is determined locally and since process scheduling is unpredictable in JCSP, it is reasonable to assume that a number of unprioritised SITE processes will be scheduled before the prioritised one. These processes will initially select 'pass' events to synchronise on. Eventually some of these 'pass' events will complete. However once the prioritised SITE process is scheduled it immediately selects the 'tock' event and steals any other processes waiting for other events. Thus, an unpredictable

(possibly random) number of processes will complete ‘pass’ events before all processes are made to synchronise on the ‘tock’ event.

Using partial priority in this way may be another way in which starvation can be avoided in otherwise priority free external choices. It may or may not be the case that using this approach will have a predictable effect on the probability of certain events being selected.

8.1.3. Modelling in CSP

While it is possible to provisionally assert that the *AltableBarrier* algorithm is deadlock free given the stress tests run on it, it is not possible to guarantee this until the algorithm has been modelled in CSP. At the time of writing no such CSP models have been attempted.

Despite this (and the relative complexity of the algorithm) modelling the *AltableBarrier* algorithm in CSP should not be considered intractable. Two different approaches to modelling the algorithm may be attempted. The first is to model the algorithm in detail, this would almost certainly require modelling individual fields as separate processes. The second is to strip the algorithm down to its barest essentials (more or less a model of the 3 phase commit protocol [10]) and identify the circumstances where such a simple system could deadlock. The rest of the verification process would then consist of proving that such circumstances are impossible (this may or may not be done using CSP).

9. Conclusion

The AltableBarriers algorithm presented in this paper, although noticeably slower than using the existing JCSP *AltingBarrier* class, can be practically applied to the prioritisation of multiway synchronisation. This allows large, infrequently triggered barrier events with large sets of enrolled processes to be consistently selected over smaller barrier events as well as channel communications without any major changes to existing JCSP classes. As such AltableBarriers are applicable in such problems as graceful termination as well as atomically pausing entire process networks.

By allowing multiway synchronisations to be prioritised, it is no longer the case that events with small sets of enrolled processes are automatically favoured over events with large sets. Further the ability to create groups of events with no internal priority within larger priority structures allows the programmer to avoid priority conflicts.

While as yet untested, there also appears to be no reason not to avoid possible problems of starvation. Partial priority as well as fair alting provide mechanisms for ensuring a degree of fairness in otherwise priority free arbitrary selections.

Acknowledgements

The comments of the anonymous reviewers on this paper are gratefully appreciated. Credit is also due to Peter Welch and Fred Barnes (and to CPA’s contributors in general) whose collective musings on the subject have helped to shape this research. This work is part of the CoSMoS project, funded by EPSRC grant EP/E053505/1.

References

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

[2] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997. ISBN: 0-13-674409-5.

[3] A.A. McEwan. Concurrent program development, d.phil thesis. *The University of Oxford*, 2006.

[4] P.H. Welch, N.C.C. Brown, J. Moores, K. Chalmers, and B. Sputh. Alting barriers: synchronisation with choice in Java using CSP. *Concurrency and Computation: Practice and Experience*, 22:1049–1062, 2010.

[5] P.H. Welch, N.C.C. Brown, J. Moores, K. Chalmers, and B. Sputh. Integrating and Extending JCSP. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch, editors, *Communicating Process Architectures 2007*, volume 65 of *Concurrent Systems Engineering Series*, pages 349–370, Amsterdam, The Netherlands, July 2007. IOS Press. ISBN: 978-1-58603-767-3.

[6] P.H. Welch and P.D. Austin. The JCSP (CSP for Java) Home Page, 1999. *Accessed 1st. May, 2011:* `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`.

[7] P.H. Welch and F.R.M. Barnes. Communicating mobile processes: introducing occam-pi. In A.E. Abdallah, C.B. Jones, and J.W. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005.

[8] P.H. Welch and F.R.M. Barnes. Mobile Barriers for occam-pi: Semantics, Implementation and Application. In J.F. Broenink, H.W. Roebbers, J.P.E. Sunter, P.H. Welch, and D.C. Wood, editors, *Communicating Process Architectures 2005*, volume 63 of *Concurrent Systems Engineering Series*, pages 289–316, Amsterdam, The Netherlands, September 2005. IOS Press. ISBN: 1-58603-561-4.

[9] C. Mohan and B. Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. *ACM SIGOPS Operating Systems Review*, 19(2):40–52, 1985.

[10] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transactions On Software Engineering*, SE-9:219–228, 1983.

[11] P.H. Welch, F.R.M. Barnes, and F.A.C. Polack. Communicating complex systems. In M.G. Hinchey, editor, *Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS-2006)*, pages 107–117, Stanford, California, August 2006. IEEE. ISBN: 0-7695-2530-X.

[12] C.J. Fidge. A formal definition of priority in csp. *ACM Transactions on Programming Languages*, Vol 15. No 4:681–705, 1993.

[13] G. Lowe. Extending csp with tests for availability. *Communicating Process Architectures*, pages 325–347, 2009.

[14] D.N. Warren. PCOMS source code. *Accessed 1st. May, 2011:* `http://projects.cs.kent.ac.uk/projects/jcsp/svn/jcsp/branches/dnw3_altbar/src/org/jcsp/lang/`.

[15] D.N. Warren. PCOMS test code. *Accessed 1st. May, 2011:* `http://projects.cs.kent.ac.uk/projects/jcsp/svn/jcsp/branches/dnw3_altbar/src/org/jcsp/demos/altableBarriers/`.