CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; **22**:1049–1062
Published online 1 March 2010 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1471

Alting barriers: synchronisation with choice in Java using JCSP[‡]

Peter Welch[1,*,†], Neil Brown[1], James Moores[2], Kevin Chalmers[3]and Bernhard Sputh[4]

[1]*Computing Laboratory*, *University of Kent*, *Canterbury*, *Kent*, *CT2 7NF*, *U.K.*
[2]*23 Tunnel Avenue*, *London SE10 0SF*, *U.K.*
[3]*School of Computing*, *Napier University*, *Edinburgh*, *EH10 5DT*, *U.K.*
[4]*Department of Engineering*, *University of Aberdeen*, *Scotland*, *AB24 3UE*, *U.K.*

SUMMARY

***Communicating Sequential Processes for Java* (JCSP) is a mature library that implements CSP-derived concurrency primitives in Java. A JCSP system is a hierarchical network of autonomous processes communicating over synchronous (optionally buffered) channels, and multiway synchronising through barriers. This paper presents a significant extension to the barrier mechanism: the fast resolution of *choice* between any number of barrier events, channel communications (in either direction) and timeouts. Previously, and in line with all currently released libraries and languages offering the CSP concurrency model, choice was restricted to channel inputs and timeouts. The paper demonstrates an application of *alting barriers* and explains the mechanisms used in their implementation that enables their use as *guards* in a choice. It also shows how choice over channel *outputs* becomes possible, as a simple consequence of having choice over barriers. Finally, an efficient implementation of CSP's *broadcasting* channels is presented (using a *phased barrier synchronisation* pattern) and a mechanism for allowing choice over these is discussed. With this extended JCSP, almost all CSP-specified systems can now be directly implemented. The new library is available under LGPL open source. Copyright © 2010 PH Welch, NCC Brown, J Moores, KV Chalmers & B Sputh.**

Received 18 May 2009; Accepted 30 May 2009

KEY WORDS: JCSP; CSP; choice; alting barriers; output guards; broadcast channels

*Correspondence to: Peter Welch, Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, U.K.
†E-mail: p.h.welch@kent.ac.uk
‡Revised version of Welch PH, Brown NCC, Moores J, Chalmers KV, Sputh B. Integrating and extending JCSP. In McEwan AA, Schneider S, Ifill W, Welch PH (eds). Communicating Process Architectures 2007. IOS Press: Amsterdam. Published with permission from IOS Press.

Contract/grant sponsor: UK EPSRC; contract/grant number: EP/E049419/1

1. INTRODUCTION

Communicating Sequential Processes for Java[§] (JSCP) [1–3] is a library of Java packages providing a concurrency model that is a judicious combination of ideas from Hoare's CSP [4,5] and Milner's π-calculus [6]. It follows many of the principles of the occam-π [7–9] programming language, exchanging compiler enforced security for programmer checked rules, losing some ultra-low process management overheads but allowing the model to be used in a mainstream programming language. Along with CTJ [10], JCSP has been used as a template for similar libraries in other environments—such as C++CSP [11], CTC++ [12] and the .NET CSP implementations [13,14].

JCSP enables the dynamic and hierarchic construction of process networks, connected by and synchronising upon a small set of primitives—such as message-passing channels and multiway events. Each process manages its own state and engages in patterns of communication with its environment (represented by channels, barriers, etc.) that can be formally specified (in CSP). Each process is independently constructed and tested without concern for multiprocessing side-effects—there is no need for locking mechanisms. In this way, long developed skills for *sequential* design and programming transfer directly into *concurrent* design and programming. Whole system (multiprocessing) behaviour yields no surprises and can be analysed for bad behaviour (e.g. deadlock) formally, with the option of assistance from automated model checkers (such as FDR [15]). The model works unchanged whether the concurrency is *internal* to a single machine (including multicore architectures) or *distributed* across many machines (including workstation clusters and the Internet).

JCSP provides an alternative concurrency model to the threads and monitor mechanisms built into Java. It is also compatible with it—indeed, it is currently implemented on top of it! With care, the two models can profitably be mixed. For example, the *straightforward* sharing of a resource can be safely managed using Java `synchronised` blocks to serialise access—rather than accept the overheads of a very simple server process. For more sophisticated management, a process should be employed—using and reasoning about an object's `wait`, `notify` and `notifyAll` methods is very hard and not necessary with JCSP. Java 1.5 includes a whole new set of concurrency primitives—some at a very low level (e.g. the atomic swaps and counts). These also provide an alternative to threads and monitors. Depending on the relative overheads between the 1.5 and classical methods, it may be worthwhile re-implementing JCSP on the lowest level 1.5 primitives. Meanwhile, we are confident in the current implementation, which has been formalised and model checked [16].

This paper presents *alting barriers* that are completely new for JCSP, together with implementation details. We also show how these facilitate channels that allow *output guards* and *broadcasting* to be used in external choice (*alting*). This paper is based upon [17], which introduced the revised APIs used here and detailed some other new extensions (*extended rendezvous* and *poison*) not reported here.

2. ALTING BARRIERS

JCSP has long provided a `Barrier` class, on which multiple processes can be enrolled. When *one* process attempts to synchronise on a barrier, it blocks until *all* enrolled processes do the same thing. When the last arrives at the barrier, all processes are released.

[§]Java is a trademark of Sun Microsystems.

A barrier corresponds to fundamental multiway *event synchronisation* in CSP. However, although CSP allows processes to offer multiway events as part of an *external choice*, JCSP does not permit this for `Barrier` synchronisation. Once a process offers to synchronise with a `Barrier`, it cannot *back off*—i.e. withdraw that offer because some other offer it made (e.g. a timeout, a channel communication or another barrier) is accepted. The reason is the same as why *channel output guards* are not allowed. Only *one* party to any synchronisation is allowed to withdraw (i.e. to use that synchronisation as a guard in external choice—*alting*). This enables event choice to be implemented with a simple (and fast) handshake from the party making the choice to its chosen partner (who is committed to waiting).

Relaxing this constraint implies resolving a choice on which all parties must agree and from which anyone can change their mind (after initially indicating approval). In [18], the first formal model of a solution is presented—a *two-phase commit* protocol employing only channel communication with no output guards—together with verification of its full equivalence under *failures-divergences* semantics. Variants of this protocol also appear in [19–22]. This solution carries a computational cost when contention is heavy and resolutions break down in their second phase and need restarting.

The original constraint had been applied in all practical CSP implementations to date. It meant that CSP systems involving external choice over multiway events could not be directly executed using the provided primitives. Instead, those systems had to be transformed (as in [18]) to meet this constraint—which meant adding the processes, channels and logic to manage the two-phase commits.

We have introduced to JCSP the `AltingBarrier` class that removes this constraint, allowing multiple barriers to be included in the guards of an `Alternative`—along with skips, timeouts, channel communications and *call channel accepts*. Currently, this is supported only for a single JVM (which can be running on a multicore processor). It uses a *fast* implementation that is not a two-phase commit, with overheads that are linear with respect to the number of events offered. This is based on the *oracle* mechanism [23,24], described in Section 2.5. Resolution of choice never breaks down but stops as soon as *one* is found, so some potential paths may never be exercised (which is, of course, perfectly legal). We believe this oracle implements a failures-divergences *refinement* of systems containing arbitrary patterns of CSP external choice between hidden multiway events (where the hiding may introduce non-determinism). We note that this solution is not compositional, but that it works for complete systems. Formal verification has not been attempted at this time.

Barriers also allow dynamic enrolment and resignation, following mechanisms introduced into occam-π [8,25]. Initial process enrolment is automatic: with the creation of new *front-ends* (see Sections 2.1 and 2.5.1) to a barrier, the enrolment count increments which means that the barrier cannot be completed (by the existing enrolled processes) until new processes take ownership of the new front-ends and offer to synchronise. Unlike occam-π, a process must explicitly resign from its barriers before termination (unless all enrolled processes terminate in the same barrier cycle). Like occam-π, a process may explicitly resign from a barrier at any time and re-enrol later (with the help of an already enrolled process to ensure that it comes back in the correct *phase*—see [25]).

2.1. User view of alting barriers

An *alting* barrier is represented by a family of `AltingBarrier` *front-ends*. Each process using the barrier must do so via its own front-end—in the same way that a process uses a channel via its channel-end. A new alting barrier is created by the `static create` method, which returns an array of front-ends—one for each enrolled process. If additional processes need later to be enrolled, further front-ends may be made from an existing one (through `expand` and `contract` methods). As with the earlier `Barrier` class, processes may temporarily `resign` from a barrier and, later, re-`enrol`.

To use this barrier, a process simply includes its given `AltingBarrier` front-end in a `Guard` array associated with an `Alternative`. Its index will be selected if and only if all parties (processes) to the barrier similarly select it (using their own front-ends).

If a process wishes to commit to this barrier (i.e. not offer it as a choice in an `Alternative`), it may `sync()` on it. However, if all parties only do this, a *non-alting* `Barrier` would be more efficient. A further shortcut (over using an `Alternative`) is provided to poll (with timeout) this barrier.

An `AltingBarrier` front-end may only be used by one process at a time (and this is checked at run-time). A process may communicate a *non-resigned* front-end to another process; but the receiving process must *mark* it before using it and, of course, the sending process must not continue to use it. If a process terminates holding a front-end, it may be recycled for use by another process via a *reset*.

Full details of expanding/contracting the set of front-ends, temporary resignation and re-enrolment, communication, marking and resetting of front-ends, committed synchronisation and time-limited polling are given in the online JCSP documentation [26].

2.2. Priorities

JCSP allows for local priorities between different channel guards in a choice. When multiple channels are ready, the guard with the highest priority is chosen. These do not—*and cannot*—apply to selection between barriers. The `priSelect()` method works locally for the process making the offer. If this were allowed, one process might offer barrier `x` with higher priority than barrier `y` ... and another process might offer them with its priorities the other way around. In which case, it would be impossible to resolve a choice in favour of `x` or `y` in any way that satisfied the conflicting priorities of both processes.

However, the `priSelect()` method is allowed for choices including barrier guards. It honours the respective priorities defined between non-barrier guards, and those between a barrier guard and non-barrier guards (which guarantees, for example, immediate response to a timeout from ever-active barriers). Relative priorities between barrier guards are *inoperative*.

2.3. Misuse

The implementation defends against misuse, throwing an `AltingBarrierError` when any of the following happens:

- different threads trying to operate on the same front-end;
- attempt to enrol while enrolled;
- attempt to use as a guard, sync, resign, expand, contract or mark while resigned;
- attempt to contract with an array of front-ends not supplied by expand.

Again, we refer to the documentation, [26], for further details and explanation.

2.4. Example

Here is a simple gadget with two modes of operation, switched by a click event (operated externally by a button in the application described below). Initially, it is in individual mode—represented here by incrementing a number and outputting it (as a string to change the label on its controlling button) as often as it can. Its other mode is group, in which it can only work if all associated gadgets are also in this mode. Group work consists of a single decrement and output of the number (to its button's label). It performs group work as often as the group will allow (i.e. until it, or one of its partner gadgets, is clicked back to individual mode).

```
import org.jcsp.lang.*;

public class Gadget implements CSProcess {

  private final AltingChannelInput click;
  private final AltingBarrier group;
  private final ChannelOutput configure;

  public Gadget (
    AltingChannelInput click, AltingBarrier group, ChannelOutput configure
  ) {
    this.click = click;
    this.group = group;
    this.configure = configure;
  }

  public void run () {

    final Alternative clickGroup = new Alternative (new Guard[] {click, group});

    final int CLICK = 0, GROUP = 1;              // indices to the Guard array

    int n = 0;
    configure.write (String.valueOf (n));

    while (true) {
      configure.write (Color.green)              // indicate mode change

      while (!click.pending ()) {                // individual work mode
        n++;                                     // work on our own
        configure.write (String.valueOf (n));    // work on our own
      }
      click.read ();                             // must consume the click

      configure.write (Color.red);               // indicate mode change
      boolean group = true;                     // group work mode
      while (group) {
        switch (clickGroup.priSelect ()) {      // offer to work with the group
          case CLICK:
            click.read ();                      // must consume the click
            group = false;                      // back to individual work mode
          break;
          case GROUP:
            n--;                                // work with the group
            configure.write (String.valueOf (n)); // work with the group
          break;
        }
      }
    }

  }
}
```

The front-end to the alting barrier shared by other gadgets in our group is given by the `group` parameter of the constructor, along with `click` and `configure` channels from and to our button process.

Note that in the above—and for most uses of these alting barriers—no methods are explicitly invoked. Just having the barrier in the guard set of the `Alternative` is sufficient.

This gadget's offer to work with the group is made by the `priSelect()` call on `clickGroup`. If all other gadgets in our group make that offer before a mouse click on our button, this gadget (together with all those other gadgets) proceed together on their joint work—represented here by decrementing the count on its button's label. All gadgets then make another offer to work together.

This sequence gets interrupted if any button on any gadget gets clicked. The relevant gadget process receives the click signal and will accept it in preference to further group synchronisation. The clicked gadget reverts to its *individual* mode of work (incrementing the count on its button's label), until that button gets clicked again—when it will attempt to rejoin the group. While any gadget is working on its own, no group work can proceed.

Here is the complete code for a system of buttons and gadgets, synchronised by an *alting barrier*. Note that this *single* event needs an *array* of `AltingBarrier` front-ends to operate—one for each gadget:

```
import org.jcsp.lang.*;

public class GadgetDemo {

  public static void main (String[] argv) {

    final int nUnits = 8;
    final One2OneChannel[] event = Channel.one2oneArray (nUnits);
    final One2OneChannel[] configure = Channel.one2oneArray (nUnits);
```

```
      final boolean horizontal = true;
      final FramedButtonArray buttons =
        new FramedButtonArray (
          "AltingBarrier: GadgetDemo", nUnits, 120, nUnits*100,
           horizontal, configure, event
        );

      // construct an array of front-ends to a single alting barrier:
      final AltingBarrier[] group = AltingBarrier.create (nUnits);

      // make the gadgets:
      final Gadget[] gadgets = new Gadget[nUnits];
      for (int i = 0; i < gadgets.length; i++) {
        gadgets[i] = new Gadget (event[i], group[i], configure[i]);
      }

      // run everything:
      new Parallel (
        new CSProcess[] {
          buttons, new Parallel (gadgets)
        }
      ).run ();

    }

  }
```

This example only contains a single alting barrier. The JCSP documentation [26] provides many more examples—including systems with intersecting sets of processes offering multiple multiway barrier synchronisations (one for each set to which they belong), together with timeouts and ordinary channel communications. There are also some *games*!

2.5. Oracle implementation

A fast resolution mechanism of choice between multiple multiway synchronisations (only) was developed originally for complex modelling applications using occam-π [23,24]. This uses an 'oracle' server process that maintains information for each barrier and (application) process enrolled. A process sends the oracle *atomically* a set of barriers with which it is prepared to engage and blocks until the oracle tells it which one has been breached. The oracle simply keeps counts of, and records, all the offer sets as they arrive. If a count for a particular barrier becomes complete (i.e. all enrolled processes have made an offer), it informs the lucky waiting processes and *atomically* withdraws all their other offers—*before* considering any new offers. No *two-phase* commit protocol is needed.

2.5.1. *Adapting the oracle for JCSP (and occam-π)*

For JCSP, these mechanics must be adapted to allow processes to make offers to synchronise that include *all* varieties of `Guard`—not just `AltingBarriers`. The central oracle process is removed and its logic and data structures distributed to the *alting barrier* structures. These now have the information to work with the usual *enable/disable* sequences implementing the `select` methods invoked on `Alternative`. Note the techniques used here for JCSP carry over to a similar notion of alting barriers for an extended occam-π [27].

The `AltingBarrier.create(n)` method first constructs a hidden *base* object—the actual alting barrier—before constructing and returning an array of `AltingBarrier` front-ends. These front-ends reference the base and are chained together. The base object is not shown to JCSP users and holds the first link to the chain of front-ends. It maintains the number of front-ends issued (which it assumes equals the number of processes currently enrolled) and a countdown of how many offers have *not* yet been made to synchronise. It has methods to expand and contract the number of front-ends and manage the resignation and re-enrolment of processes. Crucially, it implements the methods for *enabling* (i.e. receiving an offer to synchronise) and *disabling* (i.e. answering an enquiry as to whether the synchronisation has completed and, if not, withdrawing the offer). These responsibilities are delegated to it from the front-end objects.

Each `AltingBarrier` front-end maintains knowledge of the process using it (thread identifier and resigned status) and checks that it is being operated correctly. If all is well, it claims the monitor lock on the base object and delegates the methods. While holding the lock, it maintains a reference to the `Alternative` object of its operating process (which might otherwise be used by another process, via the base object, upon a successful completion of the barrier).

The oracle logic works because each full offer set from a process is handled atomically. The *select* methods of `Alternative` make individual offers (*enables*) from its guard array in sequence. A global lock, therefore, must be obtained and held throughout any enable sequence involving an `AltingBarrier`—to ensure that the processing of its set of offers (on `AltingBarriers`) are not interleaved with those from any other set. If the *enables* all fail, the lock must be released before the *alting* process blocks. If an offer (*enable*) succeeds in completing one of the barriers in the guard set, the lock must continue to be held throughout the subsequent *disable* (i.e. withdraw) sequence *and* the disable sequences of all the other partners in the successful barrier (which will be scheduled by the successful *enable*). This means that multiple processes will need to hold the lock in parallel, so that a counting semaphore (rather than monitor) has to be employed. Other disable sequences (i.e. those triggered by a successful non-barrier synchronisation) do not need to acquire this lock—even if an alting barrier is one of the guards to be disabled.

2.5.2. *Distributing the oracle*

The current JCSP release supports `AltingBarriers` only *within* a single JVM. Extending this to support them across a distributed system has some issues.

A simple solution would be to install an actual oracle process at a network location known to all. At the start of any *enable* sequence, a network-wide lock on the oracle is obtained (simply by communicating with it on a shared claim channel). Each *enable/disable* then becomes a communication to and from the oracle. The network lock is released following the same rules outlined for the single JVM (two paragraphs back). However, the network overheads for this (per *enable/disable*) and the length of time required to hold the network-wide lock look bad.

A better solution may be to operate the fast oracle logic locally within each JVM—except that, when a local barrier is potentially overcome (because all local processes have offered to engage with it), the local JCSP kernel negotiates with its partner nodes through a suitable two-phase commit protocol. This allows the local kernel to cancel safely any network offer, should local circumstances change. Only if the network negotiation succeeds are the local processes informed.

2.5.3. Take care

The logic required for correct implementation of external choice is not simple. Our original implementation contained a *race hazard* that did not strike for two years after release. To deal with this, a formal (CSP) model of Java synchronisation (including `wait`, `notify` and `notifyAll`) was built and the implementation model checked against a direct expression of external choice. That immediately confirmed the error in the original implementation *and* verified that 'corrected' version was, indeed, correct [16]. Such model checking has *not* yet been performed on the implementation of `Alternative` for multiway barriers; stress testing, however, has found no faults so far.

3. OUTPUT GUARDS

It has long been a constraint of occam-π and its derivative frameworks (e.g. JCSP, C++CSP, the CSP implementations for .NET) that channels only support input guards for use in alternatives, and not output guards. While output guards are straightforward with committed input, allowing input and output guards on the *same channel* is difficult, and thus has always been forbidden. The decision allows a much faster and simpler implementation for the languages/frameworks [23].

Now, however, alting barriers provide a mechanism on which channels with both input and output guards can easily be built, as described in [18]. Because there are still extra run-time costs, JCSP offers a *different* channel class for this—for the moment christened `One2OneChannelSymmetric`.

This *symmetric* channel is composed of two internal synchronisation objects: a standard one-to-one channel and an alting barrier. Supporting this, a new channel-end interface (actually abstract class), `AltingChannelOutput`, has been added and derives simply from `Guard` and `ChannelOutput`. Only zero-buffered one-to-one symmetric alting channels are provided for the moment.

The reading and writing processes are the only two enrolled on the channel's internal barrier—on which, of course, they can *alt*.

For any *committed* communication, a process first commits to synchronise on the internal barrier. When/if that synchronisation completes, the real communication proceeds on the internal one-to-one channel as normal.

If either process wants to use the channel as a guard in an alternative, it *offers* to synchronise on the internal barrier—an offer that can be withdrawn if one of the other guards fires first. If its offer succeeds, the real communication proceeds on the internal channel as before.

Of course, all these actions are invisible to the processes that are using the channel. The processes use the standard API for obtaining channel-ends and reading and writing. Either channel-end can be included in a set of guards for an `Alternative`.

Here is a pathological example of its use. There are two processes, A and B, connected by two opposite direction channels, c and d. From time to time, each process offers to communicate on both its channels (i.e. an offer to read and an offer to write). They do no other communication on those channels. What must happen is that the processes resolve their choices in compatible ways—one must do the writing and the other the reading. This is, indeed, what happens. Here is the A process:

```
class A implements CSProcess {

  private final AltingChannelInput in;
  private final AltingChannelOutput out;

  public A (AltingChannelInput in, AltingChannelOutput out) {
    this.in = in;
    this.out = out;
  }

  public void run () {
    final Alternative alt = new Alternative (new Guard[] {in , out});
    final int IN = 0, OUT = 1;
    ...  other local declarations and initialisation
    while (running) {
      ...  set up outData
      switch (alt.fairSelect ()) {
        case IN:
          inData = (InDataType) in.read ();
          ...  reaction to this input
        break;
        case OUT:
          out.write (outData);
          ...  reaction to this output
        break;
      }
    }
  }

}
```

The B process is the same, but with different initialisation and reaction codes and types. The system must be connected with *symmetric* channels:

```
public class PathologicalDemo {

  public static void main (String[] argv) {

    final One2OneChannelSymmetric c = Channel.one2oneSymmetric ();
    final One2OneChannelSymmetric d = Channel.one2oneSymmetric ();
    new Parallel (
      new CSProcess[] {
        new A (c.in (), d.out ()),
        new B (d.in (), c.out ())
      }
    ).run ();

  }

}
```

4. BROADCAST CHANNELS

Primitive events in CSP may synchronise many processes. Channel communications are just events and CSP permits any number of readers and writers. Many readers implies that all readers receive the same message: either all receive or none receive—this is multiway synchronisation. Many writers is a little odd: all must write the same message or no write can occur—still multiway synchronisation.

All channels currently in JCSP restrict communications to point-to-point message transfers between one writer and one reader. The `Any` channels allow any number of writers and/or readers, but only one of each can engage in any individual communication.

Allowing CSP *many-reader* (broadcasting) channels turns out to be trivial—so we may as well introduce them. The only interesting part is making them as efficient as possible.

One way is to use a process similar to `DynamicDelta` from `org.jcsp.plugNplay`. This repeatedly waits for an input and, then, outputs *in parallel* on all output channels. That introduces detectable buffering which is easily eliminated by combining the input and outputs in an extended rendezvous. We still do not have multiway synchronisation, since the readers do not have to wait for each other to take the broadcast. This can be achieved by the *delta* process outputting twice and the readers reading twice. The first message can be `null` and is just to assemble the readers. Only when everyone has taking that is the real message sent. Getting the second message tells each reader that every reader is committed to receive. The *delta* process can even send each message *in sequence* to its output channels, reducing overheads (for unicore processors).

The above method has problems if we want to allow *alting* on the broadcast. Here is a simpler and faster algorithm that shows the power of barrier synchronisation—an obvious mechanism, in retrospect, for broadcasting!

```
public class One2ManyChannelInt

  private int hold;
  private final Barrier bar;

  public One2ManyChannelInt (final int nReaders) {
    bar = new Barrier (nReaders + 1);
  }
```

```
    public void write (int n) {   // no synchronized necessary
      hold = n;
      bar.sync ();                // wait for readers to assemble
      bar.sync ();                // wait for readers to read
    }

    public int read () {          // no synchronized necessary
      bar.sync ();                // wait for the writer and other readers
      int tmp = hold;
      bar.sync ();                // we've read it!
      return tmp;
    }

  }
```

The above *broadcasting channel* supports only a fixed number of readers and no alting. This is easy to overcome using the dynamics of an `AltingBarrier`, rather than `Barrier`—but is left for another time. For simplicity, the above code is also not *dressed* in the full JCSP mechanisms for separate channel-ends, poisoning, etc. It also carries integers. Object broadcasting channels had better be carefully used! Probably, only *immutable* objects (or clones) should be broadcast. Otherwise, the readers should only ever read, not alter, the objects they receive (and anything that they reference).

The above code uses the technique of *phased barrier synchronisation* [8,25,28]. Reader and writer processes share access to the `hold` field inside the channel. That access is controlled through phases divided by the barriers. In the first phase, only the writer process may write to `hold`. In the second, only the readers may read. Then, it is back to phase one. No locks are needed.

Most of the work is done by the first barrier, which cannot complete until all the readers and writer assemble. If this barrier were replaced by an *alting* one, that could be used to enable external choice for all readers and the writer.

Everyone is always committed to the second barrier, which will therefore complete quickly. Its only purpose is to prevent the writer exiting, coming back and overwriting `hold` before all the readers have taken the broadcast. If the first barrier were replaced by an `AltingBarrier`, the second could remain as this (faster) `Barrier`.

However, other optimisations are possible—for example, by the readers decrementing a *reader-done* count (either atomically, using the new Java 1.5 concurrency utilities, or with a standard monitor lock) and with the last reader resetting the count and releasing the writer (waiting on a 2-way `Barrier`).

5. SUMMARY AND FUTURE WORK

JCSP is a mature library for concurrency in Java, with synchronisation primitives (e.g. channels and barriers), operators (e.g. parallel and choice) and concurrency model corresponding to those of CSP. JCSP may be used without knowledge of CSP, since the rich and elegant mathematics of that process algebra is burnt into its user interface and implementation. Developers get the benefits of that mathematics (e.g. a *compositional* semantics for concurrency) simply by using it. However, when formal analysis must be performed, it provides a direct bridge between the modelling and executable code—see [20,21].

This paper has reported extensions of JCSP to allow choice over multiway synchronisations (barriers), channel outputs (as well as inputs) and broadcasting channels. The implementation is *fast* in comparison with previous techniques for resolving such choices involving *two-phased commits*: no breakdowns of resolution (requiring re-tries) can occur and the computational overhead is linear in the number of events offered. Almost all constraints on the CSP systems directly supported by JCSP have now been removed.

Two open questions remain concerning this work. The first, described in Section 2.5.2, is how to provide an *efficient* resolution of the extended choice mechanisms when the processes making them are distributed. This presents an efficiency problem (though less severe) for multicore implementations as well, since sets of offers from different processes may not be processed in parallel.

The second question concerns *prioritising* the choice between barriers. As mentioned in Section 2.2, allowing processes individually to set priorities for their offers leads to unresolvable conflicts. However, associating priority with the barriers themselves ensures a consistent rule on which all processes could agree. The current mechanism is *lazy* and selects the first barrier found with a complete set of offers—which is all that CSP requires us to do. To give prioritisation a chance, all *pending* offers must be fully processed and, then, the highest priority process with a complete set of offers (if any) can be chosen. This will require some care and further thought.

Other recent additions to JCSP include *extended rendezvous* (that simplify the capture of many useful synchronisation patterns) and *poison* (for the simple and safe termination of networks or sub-networks)—see [17]. JCSP's distributed networking infrastructure is currently being revised and a new lighter implementation, suitable for use on *small ubiquitous* platforms, will be available shortly.

The revised library is available under the LGPL open source licence—see [1].

ACKNOWLEDGEMENTS

The original development of JCSP was made by Paul Austin and Peter Welch. Further contributions came from Neil Fuller, John Foster and David Taylor. The development of the Quickstone edition of JCSP was by Jim Moores, Jo Aldous, Andrew Griffin, Daniel Evans and Peter Welch. The implementation of poison (and proof thereof) was by Bernhard Sputh and Alastair Allen. Alting barriers were designed and implemented by Peter Welch. The addition of extended rendezvous and the merging of all these strands was by Neil Brown, Peter Welch and Kevin Chalmers. Peter Welch is part funded for this work through the UK EPSRC funded CoSMoS research grant (EP/E049419/1).

The authors remain in debt to the CPA/WoTUG community for continual encouragement, feedback and criticism throughout this period. We apologise unreservedly to any individuals not named above, who have nevertheless made direct technical inputs to JCSP.

We are also very grateful to the anonymous reviewers for their constructive and helpful criticisms.

REFERENCES

1. Welch PH, Austin PD. The JCSP (CSP for Java) Home Page. 1999–2009. Available at: http://www.cs.kent.ac.uk/projects/ofa/jcsp/ [July 2009].
2. Welch PH. Process oriented design for Java: Concurrency for all. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications* (*PDPTA'2000*), Arabnia HR (ed.). CSREA Press: Las Vegas, NV, U.S.A., 2000; 51–57.

3. Welch PH, Aldous JR, Foster J. CSP networking for Java (JCSP.net). *Computational Science—ICCS 2002* (*Lecture Notes in Computer Science*, vol. 2330), Sloot PMA, Tan CJK, Dongarra JJ, Hoekstra AG (eds.). Springer: Berlin, 2002; 695–708. See also: http://www.cs.kent.ac.uk/pubs/2002/1382 [July 2009].
4. Hoare CAR. *Communicating Sequential Processes*. Prentice-Hall: Englewood Cliffs, NJ, 1985. ISBN: 0-13-153289-8.
5. Roscoe AW. *The Theory and Practice of Concurrency*. Prentice-Hall: Englewood Cliffs, NJ, 1997. ISBN: 0-13-674409-5.
6. Milner R. *Communicating and Mobile Systems*: *The Pi-Calculus*. Cambridge University Press: Cambridge, 1999. ISBN-10: 0521658691, ISBN-13: 9780521658690.
7. Welch PH, Barnes FRM. Communicating mobile processes: Introducing occam-pi. *25 Years of CSP* (*Lecture Notes in Computer Science*, vol. 3525), Abdallah AE, Jones CB, Sanders JW (eds.). Springer: Berlin, 2005; 175–210.
8. Barnes FRM, Welch PH, Sampson AT. Barrier synchronisation for occam-pi. *Proceedings of the 2005 International Conference on Parallel and Distributed Processing Techniques and Applications* (*PDPTA'05*), Arabnia HR (ed.). CSREA Press: Las Vegas, NV, U.S.A., 2005; 173–179.
9. The occam-pi Programming Language. June 2006. Available at: http://www.occam-pi.org/ [July 2009].
10. Broenink JF, Bakkers AWP, Hilderink GH. Communicating threads for Java. *Proceedings of WoTUG-22*: *Architectures, Languages and Techniques for Concurrent Systems*, Cook BM (ed.). IOS Press: Amsterdam, The Netherlands, 1999; 243–262.
11. Brown NCC. C++CSP2: A many-to-many threading model for multicore architectures. *Communicating Process Architectures 2007* (*Concurrent Systems Engineering Series*, vol. 65), McEwan AA, Schneider S, Ifill W, Welch P (eds.). IOS Press: The Netherlands, 2007.
12. Orlic B, Broenink JF. Redesign of the C++ communicating threads library for embedded control systems. *5th Progress Symposium on Embedded Systems*, Karelse FSTW (ed.). STW Technology Foundation: Utrecht, The Netherlands, 2004; 141–156.
13. Lehmberg A, Olsen MN. An introduction to CSP.NET. *Communicating Process Architectures 2006*, Barnes FRM, Kerridge JM, Welch PH (eds.). IOS Press: Amsterdam, The Netherlands, 2006; 13–30.
14. Chalmers K, Clayton S. CSP for .NET based on JCSP. *Communicating Process Architectures 2006*, Barnes FRM, Kerridge JM, Welch PH (eds.). IOS Press: Amsterdam, The Netherlands, 2006; 59–76.
15. Formal Systems (Europe) Ltd. *FDR2 User Manual*, Oxford, U.K., 2000.
16. Welch PH, Martin JMR. Formal analysis of concurrent Java systems. *Communicating Process Architectures 2000*, Welch PH, Bakkers AWP (eds.). IOS Press: Amsterdam, The Netherlands, 2000; 275–301.
17. Welch PH, Brown NCC, Moores J, Chalmers KV, Sputh B. Integrating and extending JCSP. *Proceedings of Communicating Process Architectures 2007* (*CPA-2007*) (*Concurrent Systems Engineering Series*, vol. 65), Schneider S, McEwan AA, Ifill W, Welch PH (eds.). IOS Press: Amsterdam, The Netherlands, 2007; 349–370.
18. McEwan AA. Concurrent program development. *DPhil Thesis*, The University of Oxford, 2006.
19. Woodcock JCP. Using *Circus* for safety-critical applications. *VI Brazilian Workshop on Formal Methods*. Campina Grande: Brazil, 2003; 1–15.
20. Freitas LJS, Sampaio ACA, Cavalcanti ALC. JACK: A framework for process algebra implementation in Java. *16th Brazilian Symposium on Software Engineering*, Brasil, 2002; 98–113.
21. Oliveira MVM, Cavalcanti ALC. From *Circus* to JCSP. *Sixth International Conference on Formal Engineering Methods* (*Lecture Notes in Computer Science*, vol. 3308), Davies J, Schulte W, Barnett M (eds.). Springer: Berlin, 2004; 320–340. ISBN: 3-540-23841-7.
22. Oliveira MVM. Formal derivation of state-rich reactive programs using *Circus*. *PhD Thesis*, Department of Computer Science, University of York, 2005. YCST-2006/02.
23. Welch PH. A fast resolution of choice between multiway synchronisations (Invited Talk). *Communicating Process Architectures 2006*, Barnes FRM, Kerridge JM, Welch PH (eds.). IOS Press: Amsterdam, The Netherlands, 2006; 389–390.
24. Welch PH, Barnes FRM, Polack FAC. Communicating complex systems. *Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems* (*ICECCS-2006*), Hinchey MG (ed.). Stanford, CA, 2006. IEEE. ISBN: 0-7695-2530-X.
25. Welch PH, Barnes FRM. Mobile barriers for occam-pi: Semantics, implementation and application. *Communicating Process Architectures 2005* (*Concurrent Systems Engineering Series*, vol. 63), Broenink JF, Roebbers HW, Sunter JPE, Welch PH, Wood DC (eds.). IOS Press: Amsterdam, The Netherlands, 2005; 289–316. ISBN: 1-58603-561-4.
26. Welch PH. JCSP: AltingBarrier documentation, 2006. Available at: http://www.cs.kent.ac.uk/projects/ofa/jcsp/jcsp-1.1-rc4/jcsp-doc/org/jcsp/lang/AltingBarrier.html [July 2009].
27. Barnes FRM. Compiling CSP. *Proceedings of Communicating Process Architectures 2006* (*CPA-2006*) (*Concurrent Systems Engineering Series*, vol. 64), Welch PH, Kerridge J, Barnes FRM (eds.). IOS Press: Amsterdam, The Netherlands, 2006; 377–388.
28. Ritson C, Welch PH. A process-oriented architecture for complex system modelling. *Communicating Process Architectures 2007* (*Concurrent Systems Engineering Series*, vol. 65), McEwan AA, Schneider S, Ifill W, Welch P (eds.). IOS Press: Amsterdam, The Netherlands, 2007.