Communicating Process Architectures 2007
Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch
IOS Press, 2007

© 2007 The authors and IOS Press. All rights reserved.

Integrating and Extending JCSP

Peter WELCH [a], Neil BROWN [a], James MOORES [b],
Kevin CHALMERS [c] and Bernhard SPUTH [d]
[a] *Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK.*
[b] *23 Tunnel Avenue, London, SE10 0SF, UK.*
[c] *School of Computing, Napier University, Edinburgh, EH10 5DT, UK.*
[d] *Department of Engineering, University of Aberdeen, Scotland, AB24 3UE, UK.*

Abstract. This paper presents the extended and re-integrated JCSP library of CSP packages for Java. It integrates the differing advances made by Quickstone's JCSP Network Edition and the *"core"* library maintained at Kent. A more secure API for connecting networks and manipulating channels is provided, requiring significant internal re-structuring. This mirrors developments in the occam-pi language for mandated direction specifiers on channel-ends. For JCSP, promoting the concept of channel-ends to first-class entities has both semantic benefit (the same as for occam-pi) and increased safety. Major extensions include *alting barriers* (classes supporting external choice over multiple multi-way synchronisations), channel *output guards* (straightforward once we have the alting barriers), channel *poisoning* (for the safe and simple termination of networks or sub-networks) and *extended rendezvous* on channel communications (that simplify the capture of several useful synchronisation design patterns). Almost all CSP systems can now be directly captured with the new JCSP. The new library is available under the LGPL open source license.

Keywords. JCSP, Alting Barriers, Output Guards, Extended Rendezvous, Poison

Introduction

JCSP (*Communicating Sequential Processes for Java*[1]) [1,2,3,4] is a library of Java packages providing a concurrency model that is a judicious combination of ideas from Hoare's CSP [5] and Milner's π-calculus [6]. It follows many of the principles of occam-π [7,8,9,10], exchanging compiler enforced security for programmer checked rules, losing some ultra-low process management overheads but winning the model for a mainstream programming language. Along with CTJ [11], JCSP is the forerunner of similar libraries for other environments – such as C++CSP [12], CTC++ [13] and the .NET CSP implementations [14,15].

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 contracted (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, our 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 [16]). 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).

[1] Java is a trademark of Sun Microsystems

JCSP is 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[2]. 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 [17].

JCSP was developed following WoTUG's *Java Threads Workshop* [18] in 1996. Using ideas kicked around at that workshop [19], the first library (JCSP 0.5, [20]) was designed and put together by Paul Austin, a Masters student at Kent, some time in 1997. It has been under continuous development ever since by a succession of undergraduate/Masters/PhD students (Neil Fuller, Joe Aldous, John Foster, Jim Moores, David Taylor, Andrew Griffin) together with the present authors. A major undertaking was the spin-off of *Quickstone Technologies Limited* (QTL), that crafted the JCSP Network Edition. This enables the dynamic distribution of JCSP networks across any network fabric, with no change in semantics (compared with a single JVM version) – only a change in performance and the size of the system that can be run. Sadly, QTL is no more – but its work survives and is being re-integrated with the core version (which had made several independent advances, some reported here) to form the LGPL open-source new JCSP 1.1 release.

JCSP was designed for use with anything above and including Java 1.1. This compatibility with Java 1.1 has been maintained up to the current *core* release: JCSP 1.0-rc7. Given that most modern mobile devices support at least Java 1.3, we may relax this self-imposed constraint (and start, for example, using collection classes in the revised implementation). Other new mechanisms available in Java 1.5 (e.g. *generics*) and their binding into the future of JCSP are discussed in section 6.

In section 1 of this paper, we describe and motivate small changes in API and the refactoring of the channel classes and interfaces resulting from the merger of the JCSP Network Edition and JCSP 1.0-rc7. Section 2 presents the *alting barriers* that are completely new for JCSP, together with some implementation details. Section 3 shows how these facilitate channels that allow *output guards* in external choice (*alting*). The addition of *extended rendezvous* to JCSP is given in section 4, including how this works with buffered channels of various kinds. Section 5 presents the addition of channel poisoning for the safe and simple termination of networks (or sub-networks). Finally, Section 6 considers opportunities for the future of JCSP.

1. Class Restructure

1.1. JCSP 1.0-rc7

In JCSP 1.0-rc7, there are two interfaces for channel-ends: `ChannelInput` and `ChannelOutput`. There is also the abstract class `AltingChannelInput`, which extends the abstract class `Guard`[3] and the interface `ChannelInput` and enables channels to be used as *input guards* in external choice (*alting*). All this remains in JCSP 1.1.

[2]For *straightforward* management of a shared resource, we have sometimes employed direct visibility with `synchronized` blocks to serialise access – rather than accept the overheads of a very simple server process. For more sophisticated management, we would always use a process. Using and reasoning about an object's `wait`, `notify` and `notifyAll` methods should be avoided at all costs!

[3]This defines a public *type* with a set of method headers visible and used only within `org.jcsp.lang` – sadly, Java does not permit such things in an `interface`.

JCSP 1.0-rc7 channel classes, such as `One2OneChannel`, implement the `AltingChannelInput` and `ChannelOutput` classes/interfaces and all the corresponding methods. Processes take channel-end types, such as `ChannelOutput` or `AltingChannelInput`, as arguments to their constructor. Actual channel instances are passed directly to these constructors – with Java implicitly casting them down to the expected interface types.

This structure allows misuse: a process, having been given a `ChannelInput`, can cast it to a `ChannelOutput` – and vice-versa! Such tricks do enable a channel to be used in both directions, but would probably lead to tears. They are prevented in JCSP 1.1.

Classical zero-buffered fully synchronising channels are provided along with a variety of buffered versions (blocking, overwriting, overflowing). Zero-buffered channels are implemented with a different (and faster) logic than the buffered ones. A memory inefficient feature of the JCSP 1.0-rc7 implementation is that the buffered channels *sub-class* the zero-buffered classes, although that is not relevant (or visible) to the API. So, buffered classes retain fields relevant only to the unused superclass logic. This does not happen in JCSP 1.1.

1.2. JCSP Network Edition

In the JCSP Network Edition, the channel-end interfaces and abstract classes are the same as above. There are also extended interfaces, `SharedChannelInput` and `SharedChannelOutput`, that do not reveal any extra functionality but indicate that the given channel-end can be safely shared (internally) between multiple concurrent sub-processes. Channels with unshared ends, such as `One2OneChannel`, cannot be plugged into them.

A significant change is that channels, such as `One2OneChannel` and `Any2OneChannel`, are now *interfaces* (not *classes*) with two methods: `in()` for extracting the reading-end and `out()` for the writing-end. Implementations of these channel-end interfaces are package-only known classes returned by *static* methods of the `Channel` class (or actual instances of *class factories*, such as `StandardChannelFactory`).

In fact, those package-only known channel-end implementing classes are the same as the package-only known classes implementing channels – so, processes can still cast channel inputs to outputs and vice-versa!

1.3. JCSP 1.1

JCSP 1.1 merges the two libraries. Channel-end interfaces and abstract classes remain the same. Channels themselves are interfaces, as in the JCSP Network Edition. This time, however, channel-end implementations are package-only known classes that *delegate* their methods to *different* package-only known classes implementing the channels. Further, the input-end implementing classes are different from the output-end classes. So, input-ends and output-ends can no longer be cast into each other. Apart from this improvement in security, the change is not apparent and the API remains the same as that for JCSP Network Edition.

Users of the library are only exposed to interfaces (or abstract classes) representing the functionality of channels and channel-ends. Implementation classes are completely hidden. This also allows for easier future changes without affecting the visible API.

1.4. Using Channels from within a Process

The JCSP *process* view of its external channels is unchanged. Here is a simple, but *fair*, multiplexor:

```
public final class FairPlex implements CSProcess {

  private final AltingChannelInput[] in;
  private final ChannelOutput out;
```

```
  public FairPlex (AltingChannelInput[] in, ChannelOutput out) {
    this.in = in;
    this.out = out;
  }

  public void run () {
    final Alternative alt = new Alternative (in);
    while (true) {
      final int i = alt.fairSelect ();
      out.write (in[i].read ());
    }
  }

}
```

1.5. Building Networks of Processes

To build a network, channels must be constructed and used to wire together (concurrently running) process instances. In JCSP 1.0-rc7, channels were directly plugged into processes. Now, as in occam-π and the JCSP Network Edition, we must specify which *ends* of each channel to use.

All channels are now constructed using `static` methods of the `Channel` class (*or* an instance of one the specialist channel factories):

```
final One2OneChannel[] a = Channel.one2oneArray (N);   // an array of N channels
final One2OneChannel b = Channel.one2one ();           // a single channel
```

Here is a network consisting of an array of `Generator` processes, whose outputs are multiplexed through `Fairplex` to a `Consumer` process[4]. They are connected using the above channels:

```
final Generator[] generators = new Generator[N];
for (int i = 0; i < N; i++) {
  generators[i] = new Generator (i, a[i].out ());
}

final FairPlex plex = new FairPlex (Channel.getInputArray (a), b.out ());

final Consumer consumer = new Consumer (b.in ());

new Parallel (new CSProcess[] {new Parallel (generators), plex, consumer}).run ();
```

In JCSP 1.0-rc7, the actual channels (`a` and `b`) are passed to the process constructors. Now, we must pass the correct *ends*. The *input-end* of a channel is extracted using the `in()` method; the *output-end* using `out()`[5]. `FairPlex` needs an array of channel *input-ends*, which we could have constructed ourselves, applying `in()` to the individual channel elements. However, this is simplified through the `static` helper methods, `getInputArray()` and `getOutputArray()`, provided by the `Channel` factory.

[4]This example is to illustrate the use of channels, including channel arrays, in network construction. If we really only need fair and straightforward multiplexing of individual messages, it would be much simpler and more efficient to connect the generators directly to the consumer using a single `Any2OneChannel`.

[5]These correspond to the *direction specifiers* (`?` and `!`) mandated by occam-π. The method names `in()` and `out()` must be interpreted from the point of view of the *process* – not the *channel*. The *input-end* is the end of the channel from which a process inputs messages – *not* the end of the channel into which message are put. JCSP is a *process-oriented* model and our terms are chosen accordingly.

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. They allow dynamic enrollment and resignation, following mechanisms introduced into occam-π [8,21].

This 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 engages with a `Barrier`, it cannot back off (e.g. as a result of a timeout, an arriving channel communication or another barrier). 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 general, this requires a *two-phase commit* protocol, which is costly and difficult to get right [22].

This constraint has been universally applied in all practical CSP implementations to date. It means that CSP systems involving external choice over multiway events cannot, generally, be directly executed. Instead, those systems must be transformed (preserving their semantics) into those meeting the constraints – which means adding many processes and channels to manage the necessary two-phase commit.

JCSP 1.0-rc7 and 1.1 introduce the `AltingBarrier` class that overcomes that 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. It has overheads that are linear with respect to the number of barrier offers being made. It is based on the *Oracle* mechanism described at [23,24,25] and summarised in section 2.5.

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 for completion.

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 JCSP documentation (on-line at [26]).

2.2. Priorities

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 riled. Currently, the following bad things are prevented:

- o different threads trying to operate on the same front-end;
- o attempt to enrol whilst enrolled;
- o attempt to use as a guard whilst resigned;
- o attempt to sync, resign, expand, contract or mark whilst resigned;
- o 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 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;

    // make the buttons

    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. Implementation Oracle

A fast resolution mechanism of choice between multiple multiway synchronisations depends on an *Oracle* server process, [23,24,25]. This maintains information for each barrier and

each process enrolled. A process offers *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.

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

For JCSP, these mechanics need adapting to allow processes to make offers to synchronise that include *all* varieties of `Guard` – not just `AltingBarrier`s. The logic of the *Oracle* process is also unravelled 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 count-down 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 temporary 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 id* 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. Whilst 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 `AltingBarrier`s) 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 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*)[6]. 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 `AltingBarrier`s 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 follow-

[6]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.

ing 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 (i.e. the `Alternative` class) is not simple. The version just for channel input synchronisation required formalising and model checking before we got it right [17]. Our implementation has not (yet) been observed to break under stress testing, but we shall not feel comfortable until this has been repeated for these multiway events. Full LGPL source codes are available by request.

3. Output Guards

It has long been an accepted 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. 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 [22]. Because there are still extra run-time costs, JCSP 1.1 offers a *different* channel for this – for the moment christened `One2OneChannelSymmetric`.

This *symmetric* channel is composed of two internal synchronisation objects: one standard non-buffered one-to-one channel and one alting barrier. Supporting this, a new channel-end interface (actually abstract class), `AltingChannelOutput`, has been added and derives simply from `Guard` and `ChannelOutput`. We are only providing zero-buffered one-to-one symmetrically alting channels 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 using processes. They 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;

  ...  standard constructor


  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. Extended Rendezvous

Extended rendezvous was an idea originally introduced in occam-π [28]. After reading from a channel, a process can perform some actions *without* scheduling the writing process – *extending* the rendezvous between writer and reader. When it has finished those actions (and it can take its own time over this), it must then schedule the writer. Only the reader may perform this extension, and the writer is oblivious as to whether it happens.

Extended rendezvous is made available in JCSP through the `ChannelInput.startRead()` and `ChannelInput.endRead()` methods. The `startRead()` method starts the extended rendezvous, returning with a message when the writer sends it. The writer now remains blocked (engaged in the extended rendezvous) until, eventually, the reader invokes the `endRead()` method. They can be used in conjunction with *alternation* – following the (input) channel's selection, simply invoke `startRead()` and `endRead()` instead of the usual `read()`.

4.1. Examples – a Message Logger and Debugging GUI

Consider the (unlikely) task of tracking down an error in a JCSP system. We want to delay and/or observe values sent down a channel. We could insert a special process into the channel to manage this, but that would normally introduce buffering into the system. In turn, that changes the synchronisation behaviour of the system which could easily mask the error – especially if that error was a deadlock.

However, if the inserted process were to use extended rendezvous, we can arrange for there to be no change in the synchronisation. For example, the following *channel tapping* process might be used for this task:

```
class Tap implements CSProcess {

  private ChannelInput in;                  // from the original writer
  private ChannelOutput out;                // to the original reader
  private ChannelOutput tapOut;             // to a message logger

  ...  standard constructor

  public void run () {
    while (true) {
      Cloneable message = in.startRead ();  // start of extended rendezvous
      {
        tapOut.write (message.clone ());
        out.write (message);
      }
      in.endRead ();                        // finish of extended rendezvous
    }
  }
}
```

This process begins an extended rendezvous, copies the message to its *tapping* channel before writing it to the process for which it was originally intended. Only when this communication is complete does the extended rendezvous end. So long as the report to the message logger is guaranteed to succeed, this preserves the synchronisation between the original two processes: the original writer is released *if-and-only-if* the reader reads.

The extra code block and indentation in the above (and below) example is suggested to remind us to invoke the `endRead()` method, matching the earlier `startRead()`.

Instead of a message logger, we could install a process that generates a GUI window to display passing messages. As these message are only held during the extended rendezvous of `Tap`, that process no longer needs to *clone* its messages. For example:

```
class MessageDisplay implements CSProcess {

  private ChannelInput in;                  // from the tap process

  ...  standard constructor
```

```
  public void run () {

    while (true) {
      Object message = in.startRead ();     // start of extended rendezvous
      {
        ...  display message in a pop-up message box
        ...  only return when the user clicks OK
      }
      in.endRead ();                        // finish of extended rendezvous
    }
  }

}
```

Instead of performing communication in its extended rendezvous, the above process interacts with the user through a GUI. The rendezvous is not completed until the user has seen the data value and clicked OK. This in turn delays the tap process until the user clicks OK, which in turn prevents the original communication between the original two processes until the user has clicked OK.

The addition of these two processes has not altered the semantics of the original system – apart from giving the GUI user visibility of, and delaying ability over, communications on the tapped channel.

With trivial extra programming (e.g. writing a `null` to the tapping channel at the end of the extended rendezvous in `Tap`), the `MessageDisplay` could also clear its message box when the reader process takes the message. If this were done for all channels, a deadlocked system would show precisely where messages were stuck.

Such advanced debugging capabilities can be built entirely with the public API of JCSP. There is no need to delve into the JCSP implementation.

4.2. Rules

The `endRead()` function must be called exactly once after each call to `startRead()`. If the reader *poisons* the channel (section 5) between a `startRead()` and `endRead()`, the channel *will* be poisoned; but the current communication is deemed to have happened (which, indeed, it has) and no exception is thrown. In fact, `endRead()` will never throw a poison exception. Poison is explained in section 5.

4.3. Extended Rendezvous on Buffered Channels

Extended rendezvous and buffered channels have not previously been combined. occam-π, which introduced the extended rendezvous concept, does not support buffered channels. C++CSP originally disallowed extended rendezvous on buffered channels using a badly-designed exception[7]. To distinguish between channel-ends that did, and did not, support extended rendezvous, a more complicated type system would have been necessary. In addition to `AltingChannelInput` and `ChannelInput`, we would need `AltingExtChannelInput` and `ExtChannelInput`. Similarly, there would need to be two more classes for the shared versions.

Instead, we took the decision to allow extended rendezvous on buffered channels, thereby eliminating any divide. The semantics of extended rendezvous on a buffered channel are dependent on the semantics of the underlying buffer. The semantics for (some of) the standard buffers provided with JCSP are explained in the following sub-sections.

[7] In the new C++CSP2 [29], the classes have been restructured and the implementation is identical to the new JCSP implementation described here

4.3.1. Blocking FIFO Buffers

The reasoning behind the implemented behaviour of extended rendezvous on FIFO buffered channels with capacity `N` comes from the semantically equivalent pipeline of `N` '`id`' processes (i.e. *one-place* blocking buffers) connected by non-buffered channels. When an extended rendezvous is begun by the process reading from the buffered channel, the first available (that is, the oldest) item of data is read from the channel, *but not removed from its internal buffer*. If no item of data is available, the process must block. Data is only removed from the channel buffer when the extended rendezvous is completed. This mirrors the semantics of an extended rendezvous on the (unbuffered) output channel of the one-place buffer pipeline.

4.3.2. Overwriting (Oldest) Buffers

When full, writing to these channels does not block – instead, the new data overwrites the *oldest* data in the channel. Thus, the channel always holds the freshest available data – which is important for real-time (and other) systems.

There is no simple equivalent of such an overwriting buffer made from unbuffered channels, so we have no simple guidance for its semantics. Instead we choose to follow the principle of least surprise. As with the FIFO buffers, when an extended rendezvous begins, the least recent data item is read from the buffer but not removed. At any time, the writer writes to the buffer as normal, overwriting data when full – the first such one overwritten being the data just read. When the extended rendezvous completes, the data item is removed – *unless* that data 'slot' has indeed been overwritten. This requires the channel buffer to keep track of whether the data being read in an extended rendezvous has been overwritten or not.

An overwriting buffered channel breaks most of the synchronisation between reader and writer. The writer can always write. The reader blocks when nothing is in the channel, but otherwise obtains the latest data and must accept that some may have been missed. Extended rendezvous is meant to block the writer for a period after a reader has read its message – but the writer must never block!

The above implementation yields what should happen if the writer had come along after the extended rendezvous had completed. Since the writer's behaviour is independent from the reader in this case, we take the view that an earlier write (during the rendezvous) is a scheduling accident that should have no semantic impact – i.e. that it is proper to ignore it.

4.3.3. Zero Buffers

Extended rendezvous on a channel using a `ZeroBuffer` is, of course, identical to extended rendezvous on a normal unbuffered channel.

5. Poison and Graceful Termination

In [30], a general algorithm for the deadlock-free termination (and resetting) of CSP/**occam** networks (or sub-networks) was presented. This worked through the distribution of *poison* messages, resulting in poisoned *processes* having to take a defined set of termination actions (in addition to anything needed for process specific tidyness). This logic, though simple, was tedious to implement (e.g. in extending the channel protocol to introduce poison messages). Furthermore, the poison could not distribute against the flow of its carrying channels, so special changes had to be introduced to reach processes *upstream*.

The poison presented here applies to *channels* rather than processes – and it can spread upstream. When a channel is poisoned, any processes waiting on the channel are woken up and a poison exception thrown to each of them. All future reads/writes on the channel result in a poison exception being thrown – there is no antidote! Further attempts to poison the channel are accepted but ignored. This idea was orignally posted by Gerald Hilderink [31].

Poison is used to shutdown a process network – simply and *gracefully*, with no danger of deadlock. For example, processes can set a single poison exception catch block for the whole of their normal operation. The handler responds just by poisoning all its external channels. It doesn't matter whether any of them have already been poisoned.

Poison spreads around a process network viewed as an undirected graph, rather than trying to feed poison messages around a directed graph. These ideas have already been implemented in C++CSP, and by Sputh and Allen for JCSP itself [32]. This revised JCSP 1.1 poison builds on these experiences.

5.1. API Rationale

One option for adding poison to JCSP would have been to add poisonable channel-ends as separate additional interfaces. This would cause a doubling in the number of channel-end interfaces for JCSP. The reasoning presented in [33] still holds; a separation of poisonable and non-poisonable channel-ends in the type system would lead to complex common processes, that would need to be re-coded for each permutation of poisonable and non-poisonable channel-ends. Therefore, *all* channel-ends have `poison(strength)` methods.

Although all channel-ends have the poison methods, they do not have to be functional. Some channels do not permit poisoning – for example, the default ones: attempts to poison them are ignored.

5.2. Poison Strength

In [32], Sputh and Allen proposed the idea of two levels of poison – *local* and *global*. Channels could be constructed immune to local poison. Thus, networks could be built with sub-networks connected only by *local-immune* channels. Individual sub-networks could then be individually terminated (and replaced) by one of their components injecting *local* poison. Alternatively, the whole system could be shut down by *global* poison.

These ideas have been generalised to allow arbitrary (positive integer) levels of poison in JCSP 1.1. This allows many levels of nested sub-network to be terminated/reset at any of its levels. Poisonable channels are created with a specific level of *immunity*: they will only be poisoned with a poison whose *strength* is greater than their immunity. Poison exceptions carry the strength with which the channel has been poisoned: their handlers propagate poison with that same strength.

Channels carry the current strength of poison inside them: zero (poison-free) or greater than their immunity (poisoned). That strength can increase with subsequent poisoning, but is not allowed to decrease (with a weaker poison).

Note that using different strengths of poison can have non-deterministic results. For example, if different waves of poison, with different strengths, are propagating in parallel over part of a network whose channels are not immune, the strength of the poison exception a process receives will be scheduling dependent – which wave struck first! If a lower strength were received, it may fail to propagate that poison to some of its (more immune) channels before it terminates: without, of course, dealing with the stronger poison arriving later. Care is needed here.

5.3. Trusted and Untrusted Poisoners

Channel-ends of poisonable channels can be created specifically *without* the ability to poison (as in C++CSP [34]): attempts will be ignored (as if their underlying channel were not poisonable). Disabling poisoning at certain channel-ends of otherwise poisonable channels allows networks to be set up with trusted and untrusted poisoners. The former (e.g. a server process) has the ability to shut down the network. The latter (e.g. remote clients) receive the network poisoning but cannot initiate it.

5.4. Examples

Here is a standard *running-sum integrator* process, modified to support network shutdown after poisoning:

```
public class IntegrateInt implements CSProcess {

  private final ChannelInput in;
  private final ChannelOutput out;

  public IntegrateInt (ChannelInput in, ChannelOutput out) {
    this.in = in;
    this.out = out;
  }

  public void run () {
    try {
      int sum = 0;
      while (true) {
        sum += in.read ();
        out.write (sum);
      }
    } catch (PoisonException e) {             // poison everything
      int strength = e.getStrength ();
      out.poison (strength);
      in.poison (strength);
    }
  }
}
```

A guard for a channel is considered *ready* if the channel is poisoned. This poison will only be detected, however, if the channel is selected and the channel communication attempted. Here is a modification of the `FairPlex` process (from section 1.4) to respond suitably to poisoning. The only change is the addition of the `try/catch` block in the `run()` method:

```
public final class FairPlex implements CSProcess {

  private final AltingChannelInput[] in;
  private final ChannelOutput out;

  ...  standard constructor

  public void run () {
    try {
      final Alternative alt = new Alternative (in);
      while (true) {
        final int i = alt.fairSelect ();
        out.write (in[i].read ());
      }
    } catch (PoisonException e) {             // poison everything
      int strength = e.getStrength ();
      out.poison (strength);
      for (int i = 0; i < in.length; i++) {
        in[i].poison (strength);
      }
    }
  }
}
```

If the `out` channel is poisoned, the poison exception will be thrown on the next cycle of `FairPlex`. If any of the `in` channels is poisoned, its guard becomes *ready* straight away. This *may* be ignored if there is traffic from unpoisoned channels available and `FairPlex` will continue to operate normally.. However, the *fair* selection guarantees that no other input channel will be serviced twice before that poisoned (and ready) one. In the worst case, this will be after (`in.length` - 1) cycles. When the poisoned channel is selected, the exception is thrown.

5.5. Implementation

The central idea behind adding poison to all the existing channel algorithms is simple. Every time a channel wakes up from a `wait`, it checks to see whether the channel is poisoned. If it is, the current operation is abandoned and a `PoisonException` (carrying the poison strength) is thrown.

However, with just the above approach, it would be possible for a writing process (that was late in being rescheduled) to observe poison added by a reader *after* the write had completed successfully. This was discovered (by one of the authors [35]) from formalising and (FDR [16]) model checking this (Java) implementation against a more direct CSP model, using techniques developed from [17].

Therefore, an extra field is added so that a successfully completed communication is always recorded in the channel, regardless of any poison that may be injected afterwards. Now, the writer can complete normally and without exception – the poison remaining in the channel for next time. This correction has been model checked [35]. It has also been incorporated in the revised C++CSP [36].

6. Conclusions and Future Work

The latest developments of JCSP have integrated the JCSP Network Edition and JCSP 1.0-rc7, keeping the advances each had made separately from their common ancestor. New concepts have been added: choice between multiple multiway synchronisations (alting barriers), output guards (symmetric channels), extended rendezvous and poison. The revised library is LGPL open sourced. We are working on further re-factorings to allow third parties to add new *altable* synchronisation primitives, without needing to modify existing sources. We list here a few extensions that are have been requested by various users and are likely for future releases. Of course, with open source, we would be very pleased for others to complete these with us.

6.1. 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 cycles by waiting for an input and, then, outputting *in parallel* on all output channels. That in-

troduces detectable buffering which is easily eliminated by combining the input and outputs in an extended rendezvous (Section 4). 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 (never change) the objects they receive (and anything that they reference).

The above code uses the technique of *phased barrier synchronisation* [8,21,37]. 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's 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 cannot therefore stick. It's 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, perhaps, on a 2-way `Barrier`).

6.2. Java 1.5 Generics

Java 1.5 (also known as Java 5) was a major release that introduced many new features. The three main additions pertinent to JCSP are generics, autoboxing, and the new `java.util.concurrent` package (and its subpackages).

Generics in Java are a weak form of *generic* typing. Their primary use is to enhance semantic clarity and eliminate some explicit type casting (whilst maintaining type safety). They have been particularly successful in the revised collection classes.

Generics can be used to type more strongly JCSP channels (and avoid the cast usually needed on the return `Object` from a `read/startRead()` method). It would make the type of the channel explicit and enforced by the compiler. Generics require a Java compiler of version 1.5 or later, but they can be compiled into earlier bytecode versions executable by Java 1.3.

6.3. Java 1.5 Autoboxing

Autoboxing is the term for the automatic conversion from primitive types (such as `int` or `double`) into their class equivalents (`Integer` and `Double` respectively). Particularly when combined with generics, this allows primitive types directly to be used for communicating with generic processes through object-carrying channels. For example, if both autoboxing and generics are used in future versions of JCSP, the following codes would be legal. First, we need a generic channel:

```
One2OneChannel<Double> c = Channel.<Double>one2one (new Buffer<Double> (10));
```

Then, a writing process could execute:

```
out.write (6.7);
```

where `out` is the output-end of the above channel (i.e. `c.out()`). A reading process could execute:

```
double d = in.read ();
```

where `in` is the input-end of the above channel (i.e. `c.in()`). Note the lack of any casts in the above codes.

Like generics, autoboxing requires a 1.5 compiler but can be compiled to be executable by earlier versions, such as 1.3. This makes generics and autoboxing a potential candidate for inclusion in JCSP that would still allow Java 1.3 compatibility to be maintained – although it would mean that JCSP developers would need a Java 1.5 compiler.

6.4. Java 1.5 New Concurrency Utilities

The `java.util.concurrent` package contains new concurrency classes. Some classes complement JCSP well: the `CopyOnWriteArrayList` and `CopyOnWriteArraySet` classes can be safely shared between processes to increase efficiency.

Some classes have close similarity to certain JCSP primitives. `CyclicBarrier` is one such class, implementing a barrier (but with a useful twist in its tail). However, it does not support dynamic enrolment and resignation, nor any form of use in anything resembling *external choice*. Its support for the thread *interruption* features of Java makes it, arguably, more complex to use.

`BlockingQueue` looks similar to a FIFO-buffered channel, with `Exchanger` similar to an unbuffered channel. However, they are not direct replacements since neither class supports *external choice*.

The *atomic* classes (in `java.util.concurrent.atomic`) are tools on which JCSP primitives might profitably be built. This is an avenue for future work.

6.5. Networking

Consideration must also be taken as to how the new features in the core can be implemented into JCSP Network Edition. One of the strengths provided in JCSP is the transparency (to the process) of whether a channel is networked or local. If (generic) typed channels are to be implemented, then a method of typing *network* channels must also be available. This brings with it certain difficulties. Guarantees between two nodes must be made to ensure that the networked channel sends and receives the expected object type. However, of more importance at the moment is the implementation of networked barriers, and also networked alting barriers, to allow the same level of functionality at the network level as there is at the local level. Extended rendezvous and guarded outputs on network channels are also considerations.

If the move to exploit Java 1.5 is made in JCSP, then certain features of Java can be taken advantage of in the network stack to improve resource usage, and possibly performance. Java 1.4 introduced a form of *'channel'*, in its `java.nio.channels` package, that can be used to have the native system do some of the work for us. These channels can be used for multiplexing. Since they can represent network connections, we may be able to prune the current networking infrastructure of JCSP to reduce the number of processes needed to route things around – saving memory and run-time overheads.

Attribution

The original development of JCSP was done by Paul Austin and Peter Welch. Further contributions came from Neil Fuller, John Foster and David Taylor. The development of JCSP Network Edition was done by Jim Moores, Jo Aldous, Andrew Griffin, Daniel Evans and Peter Welch. The implementation of poison (and proof thereof) was done 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 done by Neil Brown, Peter Welch and Kevin Chalmers.

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 nevetheless made direct technical inputs to JCSP.

References

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

[2] P.H.Welch. Process Oriented Design for Java: Concurrency for All. In H.R.Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, volume 1, pages 51–57. CSREA, CSREA Press, June 2000.

[3] P.H. Welch, J.R. Aldous, and J. Foster. CSP networking for java (JCSP.net). In P.M.A. Sloot, C.J.K. Tan, J.J. Dongarra, and A.G. Hoekstra, editors, *Computational Science - ICCS 2002*, volume 2330 of *Lecture Notes in Computer Science*, pages 695–708. Springer-Verlag, April 2002. ISBN: 3-540-43593-X. See also: `http://www.cs.kent.ac.uk/pubs/2002/1382`.

[4] P.H. Welch and B. Vinter. Cluster Computing and JCSP Networking. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 213–232, IOS Press, Amsterdam, The Netherlands, September 2002. ISBN: 1-58603-268-2.

[5] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
[6] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999. ISBN-10: 0521658691, ISBN-13: 9780521658690.
[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] F.R.M. Barnes, P.H. Welch, and A.T. Sampson. Barrier synchronisation for occam-pi. In Hamid R. Arabnia, editor, *Proceedings of the 2005 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'05)*, pages 173–179, Las Vegas, Nevada, USA, June 2005. CSREA Press.
[9] F.R.M. Barnes. occam-pi: blending the best of CSP and the pi-calculus. `http://www.occam-pi.org/`, 10 February 2007.
[10] The occam-pi programming language, June 2006. Available at: `http://www.occam-pi.org/`.
[11] J.F. Broenink, A.W.P. Bakkers, and G.H. Hilderink. Communicating Threads for Java. In Barry M. Cook, editor, *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, pages 243–262, 1999.
[12] N.C.C. Brown and P.H. Welch. An Introduction to the Kent C++CSP Library. In Jan F. Broenink and Gerald H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 139–156, 2003.
[13] B. Orlic. and J.F. Broenink. Redesign of the C++ Communicating Threads Library for Embedded Control Systems. In F. Karelse STW, editor, *5th Progress Symposium on Embedded Systems*, pages 141–156, 2004.
[14] A. Lehmberg and M.N. Olsen. An Introduction to CSP.NET. In Frederick R. M. Barnes, Jon M. Kerridge, and Peter H. Welch, editors, *Communicating Process Architectures 2006*, pages 13–30, 2006.
[15] K. Chalmers and S. Clayton. CSP for .NET Based on JCSP. In Frederick R. M. Barnes, Jon M. Kerridge, and Peter H. Welch, editors, *Communicating Process Architectures 2006*, pages 59–76, 2006.
[16] Formal Systems (Europe) Ltd., 3, Alfred Street, Oxford. OX1 4EH, UK. *FDR2 User Manual*, May 2000.
[17] P.H. Welch and J.M.R. Martin. Formal Analysis of Concurrent Java Systems. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 275–301, 2000.
[18] WoTUG: Java Threads Workshop, 1996. Available at: `http://wotug.ukc.ac.uk/parallel/groups/wotug/java/`.
[19] P.H.Welch. Java Threads in the Light of occam/CSP. In P.H.Welch and A.W.P.Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications*, volume 52 of *Concurrent Systems Engineering Series*, pages 259–284, Amsterdam, April 1998. WoTUG, IOS Press.
[20] P. Austin. JCSP: Early Access, 1997. Available at: `http://www.cs.kent.ac.uk/projects/ofa/jcsp0-5/`.
[21] 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.
[22] A.A. McEwan. *Concurrent Program Development*. DPhil thesis, The University of Oxford, 2006.
[23] P.H. Welch. A Fast Resolution of Choice between Multiway Synchrnoisations (Invited Talk). In Frederick R. M. Barnes, Jon M. Kerridge, and Peter H. Welch, editors, *Communicating Process Architectures 2006*, pages 389–390, 2006.
[24] P.H. Welch, F.R.M. Barnes, and F.A.C. Polack. Communicating complex systems. In Michael 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.
[25] P.H. Welch. TUNA: Multiway Synchronisation Outputs, 2006. Available at: `http://www.cs.york.ac.uk/nature/tuna/outputs/mm-sync/`.
[26] P.H. Welch. JCSP: AltingBarrier Documentation, 2006. Available at: `http://www.cs.kent.ac.uk/projects/ofa/jcsp/jcsp1-0-rc7/jcsp-docs/jcsp/l%ang/AltingBarrier.html`.
[27] F.R.M. Barnes. Compiling CSP. In P.H. Welch, J. Kerridge, and F.R.M. Barnes, editors, *Proceedings of Communicating Process Architectures 2006 (CPA-2006)*, volume 64 of *Concurrent Systems Engineering Series*, pages 377–388. IOS Press, September 2006.
[28] F.R.M. Barnes and P.H. Welch. Prioritised Dynamic Communicating Processes - Part I. In James Pascoe, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, pages 321–352, 2002.
[29] N.C.C. Brown. C++CSP2. `http://www.cppcsp.net/`, 10 February 2007.
[30] P.H. Welch. Graceful Termination – Graceful Resetting. In *Applying Transputer-Based Parallel Machines*,

Proceedings of OUG 10, pages 310–317, Enschede, Netherlands, April 1989. Occam User Group, IOS Press, Netherlands. ISBN 90 5199 007 3.

[31] G.H. Hilderink. Poison, 2001. Available at: `http://occam-pi.org/list-archives/java-threads/msg00528.html`.

[32] B.H.C. Sputh and A.R. Allen. JCSP-Poison: Safe Termination of CSP Process Networks. In *Communicating Process Architectures 2005*, 2005.

[33] N.C.C. Brown. Rain: A New Concurrent Process-Oriented Programming Language. In Frederick R. M. Barnes, Jon M. Kerridge, and Peter H. Welch, editors, *Communicating Process Architectures 2006*, pages 237–252, 2006.

[34] N.C.C. Brown. C++CSP Networked. In Ian R. East, David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, pages 185–200, 2004.

[35] B.H.C. Sputh. *Software Defined Process Networks*. PhD thesis, University of Aberdeen, August 2006. Initial submission.

[36] N.C.C. Brown. C++CSP2: A Many-to-Many Threading Model for Multicore Architectures. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch, editors, *Communicating Process Architectures 2007*, volume 65 of *Concurrent Systems Engineering Series*, 2007.

[37] C. Ritson and P.H. Welch. A Process-Oriented Architecture for Complex System Modelling. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch, editors, *Communicating Process Architectures 2007*, volume 65 of *Concurrent Systems Engineering Series*, 2007.