Communicating Process Architectures 2000
P.H. Welch and A.W.P. Bakkers (Eds.)
IOS Press, 2000

Formal Analysis of Concurrent Java Systems

Peter H. Welch
Computing Laboratory
University of Kent at Canterbury
CT2 7NF, UK

Jeremy M.R. MARTIN
Oxford Supercomputing Centre
Wolfson Building
Parks Road
Oxford OX1 3QD, UK

Abstract. Java threads are synchronised through primitives based upon *monitor* concepts developed in the early 1970s. The semantics of Java's primitives have only been presented in natural language – this paper remedies this with a simple and formal CSP model. In view of the difficulties encountered in reasoning about any non-trivial interactions between Java threads, being able to perform that reasoning in a formal context (where careless errors can be highlighted by mechanical checks) should be a considerable confidence boost. Further, automated model-checking tools can be used to root out dangerous states (such as deadlock and livelock), find overlooked race hazards and prove equivalence between algorithms (e.g. between optimised and unoptimised versions). A case study using the CSP model to prove the correctness of the JCSP and CTJ channel implementations (which are built using standard Java monitor synchronisation) is presented. In addition, the JCSP mechanism for *ALTing* (i.e. waiting for and, then, choosing between multiple events) is verified. Given the history of erroneous implementations of this key primitive, this is a considerable relief.

Keywords: Java, threads, monitor, CSP, JCSP, CTJ, formal verification.

1 Introduction

Java has a built-in concurrency model based upon threads and monitors. It is simple to understand but very hard to apply. Its methods scale badly with complexity. Almost all Java multi-threaded codes making direct use of these primitives that we have seen (including our own) have contained race hazards – with some of our own remaining undetected for over two years (although in daily use, with their source codes on the web and their algorithms presented without demur to several Java-literate audiences). Our failures only showed themselves when faster JITs (*Just-In-Time* compilers) enabled certain threads to trip the wrong way over unspotted race hazards, corrupting some internal state that (in due course) led to deadlock. Debugging the mess was not easy – fortunately, the application was not safety-critical!

We regard this as evidence that there is something *hard* about Java multithreading. We are not alone in this opinion – numerous warnings circulate on the web (e.g. [4] from Sun's own web pages).

Java monitors, therefore, are not language elements with which we want to "think" – at least, not without some serious help. The first step in getting that help is to build a formal model that describes what is happening. The particular semantics given here is a *CSP* (*Communicating Sequential Processes*)[5] one. The importance of CSP is that it is an algebra for concurrent systems – a formal piece of mathematics with which we can specify requirements precisely (including properties like deadlock-freedom) and prove that our implementations satisfy them. Further, some powerful and mature CSP tools can be applied – for example, *FDR* (*Failures-Divergences-Refinement*) from Formal Systems Ltd.[2] and Jeremy Martin's *deadlock/sat* checker[9, 10].

There is some - but, worryingly, not widespread - concern in the Java community about the absence of such a formal model (e.g. see [12]). Without it, we will always remain uncomfortable about the security of any multithreaded product. As Tony Hoare said in his 1980 Turing Award speech[6], there are two kinds of computer systems that sell:

- those that are obviously right ...
- and those that are not obviously wrong ...

and he noted that it's much easier, of course, to produce the latter. Guess which kind we are peddling! We wonder how many surprises will pop up when we start applying CSP tools to Java codes?

This paper extends the original presentation of this model[17]. The case studies include verification of the JCSP channel implementation (Sections 3-5), the CTJ channel (Section 6) and the JCSP *ALTing* mechanism (Section 7).

Reaching the last of these three goals was the real motivation behind the development of this formal model for Java monitor operations. Although the code – two interacting monitors hit by many threads – fits on to less than two pages (see Section 7), its safety analysis repeatedly fooled professional Java experts. The original[15] JCSP implementation of *ALTing*, which had the same length as the one presented here, was declared safe – albeit with a certain amount of finger crossing! Two years later, when we were finally feeling comfortable with it, we had quite a shock when it suddenly deadlocked.

This monitor implementation of *ALTing* is not particularly lengthy or complex. Modern and near future systems will demand multithreaded code synchronisation that will be at least as difficult. Many of these systems will be safety-critical, where in-service failure costs lives. It is for such reasons that this formal model is offered.

2 The CSP Model

The key Java primitives for thread synchronisation are:

- `synchronized` methods and blocks;
- the methods `wait`, `notify` and `notifyAll` of the `Object` superclass.

Their informal (natural language) semantics will be briefly summarised as we build their CSP model. Otherwise, we assume familiarity with CSP and Java basics.

2.1 Objects and Threads

We shall model a system consisting of a set of Java objects and threads. Let *Objects* be an enumeration of all Java objects. For any particular Java application being CSP-modelled, this can be restricted to just those objects on which any threads in the application are ever `synchronized`. Usually, this will be finite and small – for example:

$$Objects = \{0, 1, 2\}$$

Let *Threads* be an enumeration of all Java threads. For any particular Java application being CSP-modelled, this can be restricted to just those threads that are created and started. Sometimes, this may be unbounded or large – for example:

$$Threads = \{0, 1\}$$

2.2 Synchronisation Events

We define a collection of channels to model Java's synchronisation events:

$$\textbf{channel}\ claim, release, waita, waitb, notify, notifyall : Objects.Threads$$

This introduces six families of channel, with each family indexed by an object *and* a thread. For example $claim.o.t$, where o is in $Objects$ and t is in $Threads$.

2.3 The User Process Interface to Java Monitors

We define the Java programmer's interface to monitors. For the moment, we'll ignore recursive locks by a particular thread on a particular object (i.e. the re-acquisition of a monitor lock by a thread that already has it). This can easily be handled by using processes to represent the relevant lock counts. Also, we will set aside the possible `InterruptedException` that may get raised by the `wait` method. Our model can be simply extended to account for this but these extensions will be reported in a later paper.

Entry and exit to a `synchronized` block or method, `o.wait()`, `o.notify()` and `o.notifyAll()` are modelled, respectively, by the following five processes:

$$\begin{aligned}
STARTSYNC(o, me) &= claim.o.me \rightarrow SKIP \\
ENDSYNC(o, me) &= release.o.me \rightarrow SKIP \\
WAIT(o, me) &= waita.o.me \rightarrow release.o.me \rightarrow \\
&\quad waitb.o.me \rightarrow claim.o.me \rightarrow \\
&\quad SKIP \\
NOTIFY(o, me) &= notify.o.me \rightarrow SKIP \\
NOTIFYALL(o, me) &= notifyAll.o.me \rightarrow SKIP
\end{aligned}$$

where me is the thread performing the action.

The interesting one is the $WAIT(o, me)$ process. The first event $(waita.o.me)$ puts its invoking thread (me) in the *wait-set* of the monitor object (o) – see Section 2.4.2. The second event $(release.o.me)$ releases the lock it was holding on the monitor object (o) – see Section 2.4.1. The third event $(waitb.o.me)$ represents its commitment to leave the *wait-set* of (o). The final event $(claim.o.me)$ is its re-acquisition of the monitor lock.

Note that this *WAIT(o,me)* process has been modified from the original version of this model [16]. At first, we had the *release* event preceding the *waita*. Subsequent FDR analysis threw up some unexpected deadlocks and we returned to the java definition document which revealed a misunderstanding in our interpretation of the natural language explanation. This has now been corrected as described above: an object needs to join the wait-set *before* releasing the monitor – otherwise it might miss being notified. Again, this shows the importance and usefulness of having a simple *formal* definition of these semantics.

2.4 Monitor Processes

Every Java object can be used as a monitor. In our model, there will be a monitor process, $MONITOR(o)$, for each o in $Objects$. This process is itself the parallel composition of two processes:

$$MONITOR(o) = MLOCK(o) \parallel MWAIT(o, \{\})$$

where $MLOCK(o)$ controls the *locking* of object o's monitor (i.e. deals with `synchronized`) and $MWAIT$ controls the (initially empty) *wait-set* of threads currently stalled on this monitor (i.e. deals with `wait`, `notify`, `notifyAll`). The alphabet of $MONITOR(o)$ is the union of its component processes, which are defined next.

2.4.1 Locking the Monitor

Each $MLOCK(o)$ process is basically a binary semaphore. Once it has been claimed by a thread (i.e. entry to a `synchronized` method or block), only a release from that *same* thread (i.e. exit from the entered `synchronized` method or block) will let it go. If this were all it had to do, it could be simply modelled by:

$$MLOCK(o) = claim.o?t \rightarrow release.o.t \rightarrow MLOCK(o)$$

$$\alpha MLOCK(o) = \{claim.o.t, release.o.t \mid t \in Threads\}$$

However, one of the constraints in Java is that an `o.wait()`, `o.notify()` or `o.notifyAll()` is only allowed if the invoking thread has the monitor lock on `o`. In Section 2.3, these invocations are modelled by (user) processes commencing, respectively, with the events $wait.o.t$, $notify.o.t$ and $notifyAll.o.t$ (where t is the invoking thread).

This constraint is enforced by including these events in the alphabet of $MLOCK(o)$, but *refusing* them in its (initial) unlocked state. In the locked state, these events are accepted but have no impact on the state:

$$\begin{aligned} MLOCK(o) &= claim.o?t \rightarrow MLOCKED(o,t) \\ MLOCKED(o,t) &= release.o.t \rightarrow MLOCK(o) \\ &\Box\ notify.o.t \rightarrow MLOCKED(o,t) \\ &\Box\ notifyall.o.t \rightarrow MLOCKED(o,t) \\ &\Box\ waita.o.t \rightarrow MLOCKED(o,t) \end{aligned}$$

$$\alpha MLOCK(o) = \left\{ \begin{array}{c} claim.o.t, release.o.t, \\ notify.o.t, notifyall.o.t, \\ waita.o.t \mid t \in Threads \end{array} \right\}$$

2.4.2 Managing the Wait-Set

The $MWAIT(o, ws)$ process controls the *wait-set* (ws) belonging to the monitor object (o). This set contains all threads that have invoked `o.wait()` but have not yet been *notified*.

New threads are added to the set via the $waita$ channel. A *notify* event results in *one* thread being non-deterministically selected from the set and reactivated – if the set is empty, the event is still accepted but nothing changes. A *notifyall* event results in *all* the waiting threads being reactivated in some non-deterministic order

$$\begin{array}{l} MWAIT(o, ws) = \\ \quad (waita.o?t \rightarrow MWAIT(o, ws \cup \{t\})) \ \Box \\ \quad \left(\begin{array}{l} notify.o?t \rightarrow \\ \quad \textbf{if}\ (|ws| > 0)\ \textbf{then} \\ \quad\quad \bigsqcap_{s \in ws} waitb.o!s \rightarrow MWAIT(o, ws - \{s\}) \\ \quad \textbf{else} \\ \quad\quad MWAIT(o, \{\}) \end{array} \right) \ \Box \\ \quad (notifyall.o?t \rightarrow RELEASE(o, ws)) \end{array}$$

and where:

$$
\begin{array}{l}
RELEASE(o, ws) = \\
\quad \textbf{if } (|ws| > 0) \textbf{ then} \\
\quad\quad \Box_{t \in ws} \; waitb.o!t \rightarrow \quad RELEASE(o, ws - \{t\})) \\
\quad \textbf{else} \\
\quad\quad MWAIT(o, \{\}) \\
\\
\alpha MWAIT(o) = \\
\quad \{ \; waita.o.t, waitb.o.t, notify.o.t, notifyall.o.t \mid t \in Threads \; \}
\end{array}
$$

2.4.3 Visualisation

One of the difficulties of working with threads and monitors is that it is hard to visualise what is happening. One of the strengths of CSP models is that they correspond to notions of hardware – *layered networks of components connected by wires* – that are easy to visualise and whose operations correspond to intuitive concepts of communication and synchronisation.

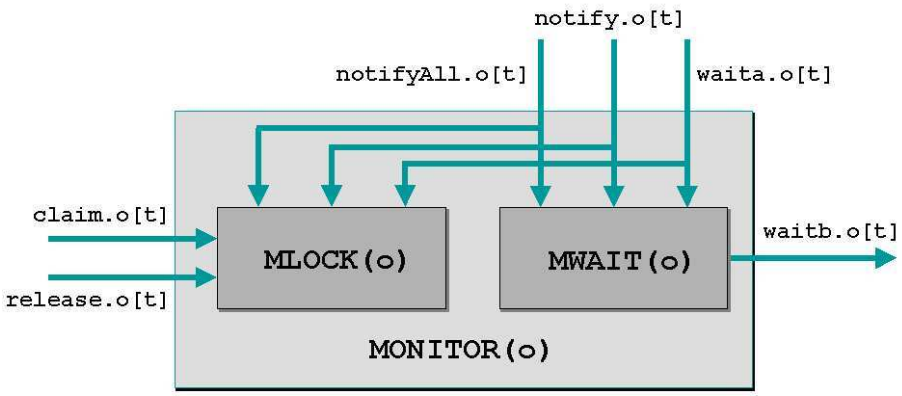

Figure 1: The Monitor Process (for Object `o`)

Figure 1 represents the CSP process enforcing the monitor rules for a Java object `o`. Each arrow represents an array of channels – one for each thread, `t`, that needs to `synchronize`, `wait` or `notify` on this monitor. The split channels are *broadcasters* – both sub-processes must input for the communication to take place.

An observation is that this is a rather specialised and complex object to be given as the *sole* primitive for synchronisation control of multithreaded systems. At least, that is in comparison with the CSP channel primitive, whose visualisation is as a bare wire!

3 A Case Study: the JCSP Channel

In Section 1, we said that we do not like to "think" at the level of Java monitors. Although the semantics of individual operations are simple enough and now formally defined, correct usage requires an understanding of how the monitor methods interact. This means that such methods cannot be designed or understood individually – their logics are very tightly coupled and all must be considered *at the same time*. This is compounded when different monitors invoke each others' methods! This approach to design does not scale well.

Formal methods certainly help but even they may become overwhelmed without some discipline that eliminates the strong coupling between threads. That discipline can be provided by CSP itself.

Java's flexibility means that we can ignore the built-in monitor model and build a high-level API to something that does scale and with which we can "think" - for instance, the occam/CSP model. JCSP[18] (*CSP-for-Java*) is a Java class library spun out from work started in the *occam-for-all* project in the Portable Software Tools for Parallel Architectures managed programme of the UK Engineering and Physical Sciences Research Council. It provides an occam3[3] concurrency framework for Java applications. CSP designs (with some occam-like caveats) can be directly crafted into Java code with no stress.

Currently, JCSP is built upon standard Java monitors and suffers (but does not increase) their overheads. Ultra-low overheads for process management, which lead to ultra-low latencies for event handling (e.g. for external message passing) are possible, although this would require building something close to (and derived from) our occam kernels into specialised JVMs. JCSP already supports shared-memory (SMP) concurrency, for which CSP primitives provide excellent control. JCSP does not currently support distributed memory architectures, although being derived from the old transputer model, of course it could.

If we can prove that the JCSP implementation of its primitives (e.g. channels) really gives us the corresponding CSP semantics, this will not only be a huge relief (since that was the original intention after all!), but it will also mean that formal analysis of JCSP designs can be done *directly* in terms of those primitives (and not on Java monitors). That will be a considerable simplification. Since CSP semantics are well-behaved under parallel composition, formal design and analysis of large multithreaded systems becomes practical. It will raise both our confidence in these systems and their real quality.

3.1 Visualisation of the Verification

3.1.1 Main Theorem

The theorem to be proved is that the system shown in Figure 2 is equivalent to the one shown in Figure 3.

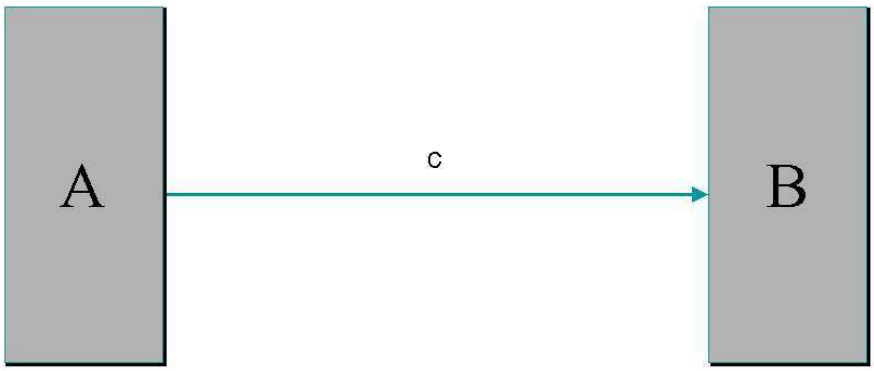

Figure 2: CSP channel communication

Figure 2 shows two processes, `A` and `B`, communicating over a CSP channel `c`. Each process guarantees that no *parallel* writes (for `A`) or reads (from `B`) are allowed. The internal channel, `c`, is hidden from the outside world.

Figure 3 shows two processes, `Aj` and `Bj`, communicating via three intermediary processes. `Aj` is the same as `A` except that the CSP output, `c ! mess`, is changed to the JCSP method invocation, `c.write (mess)`, where `c` is now a JCSP `One2OneChannel` object (see Section 3.2). The implementation of this method uses `c` as a monitor – hence the `MONITOR(c)` process. It also uses two state variables, `channel_hold` and `channel_empty`, represented by the *variable* processes `Hold(c)` and `Empty(c)`. These latter

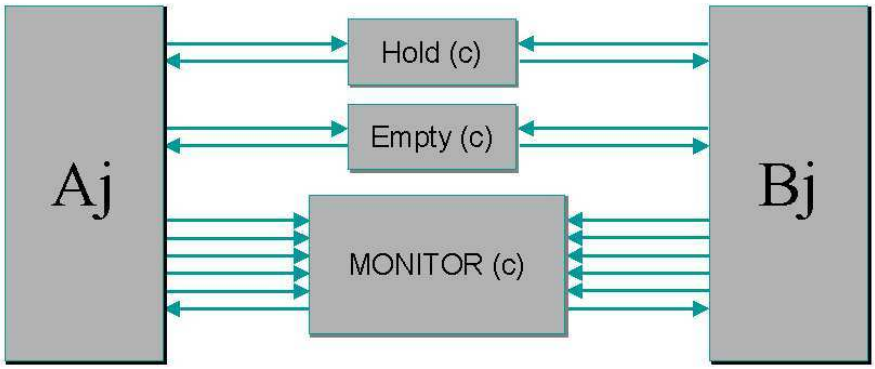

Figure 3: JCSP channel communication

processes service simple *get* and *set* channels for getting and setting values - no requests are ever refused. `Empty(c)` holds boolean values (that indicate if one side is ready to communicate) and `Hold(c)` holds the message being sent (and whose actual value is irrelevant to the operation of this channel).

`Bj` is related to `B` in the same way as `Aj` relates to `A`. The CSP input, `c ? mess`, is changed to the JCSP method invocation, `mess = c.read()`, where `c` is the same `One2OneChannel` object used by `Aj`. The implementation of this method (see Section 3.2) interacts with the same three intermediary processes as `Aj`, but uses its own sets of channels.

To prove this theorem is quite daunting and we would like some mechanical help. The FDR model checker cannot yet be employed because the behaviour of the `A`/`Aj` and `B`/`Bj` processes when they are not communicating has not been specified. Model checkers need a completely specified system into which to get their teeth.

3.1.2 Parallel Introduction

Although this may seem a strange thing to want to do, *gratuitous* introduction of parallel processes can be done anywhere.

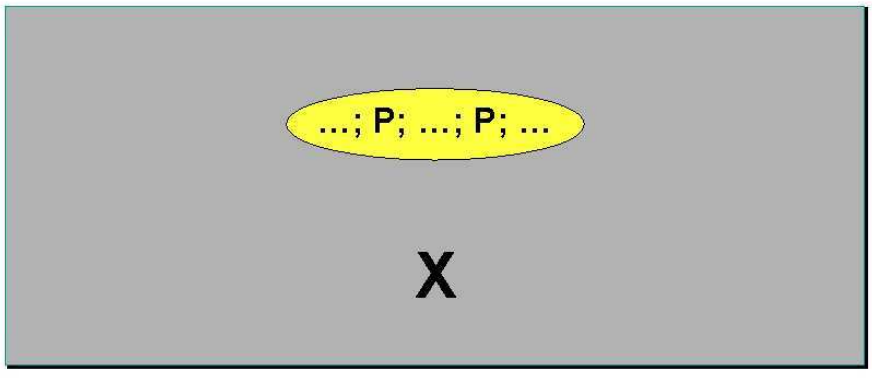

Figure 4: Do `P` Yourself

Suppose `X` contains one or more instances of a process `P` that are executed in sequence – see Figure 4. Then, `X` may be replaced by the system shown in Figure 5. `X'` is the same as `X`, except that each occurrence of `P` is replaced by synchronisation on the CSP events `ping` and `pong` (in that order). These events are in the alphabet of `Xb'`, but are *hidden* from the outside environment. `Xb'` is a *buddy* process for `X'` and is completely defined apart from `P`.

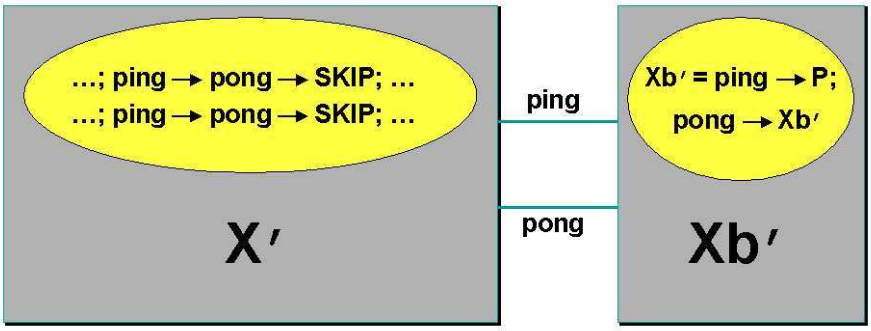

Figure 5: Delegate `P` to a Buddy

All it does is wait for a `ping`, perform `P` on behalf of its buddy and, then, let its buddy know that it's finished by synchronising on `pong`. It repeats doing this forever.

We don't need a model checker to prove the equivalence of figures 4 and 5 – it's almost a *one-liner* from the basic algebraic laws of CSP.

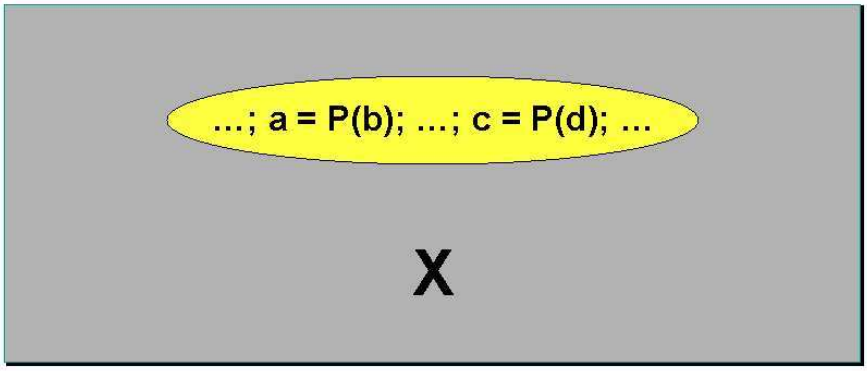

Figure 6: Do `P` (with side-effects) Yourself

Figure 6 shows a slightly more useful version of Figure 4, where the process to be delegated accesses and modifies some of the state of `X`. Further, the particular states being modified vary between instances.

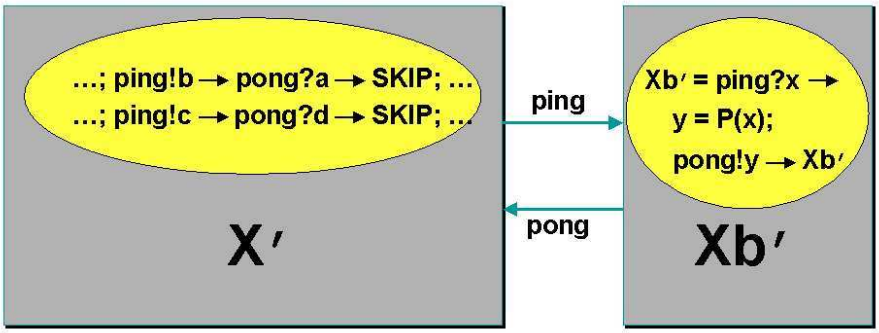

Figure 7: Delegate `P` (with side-effects) to a Buddy

In this case, the `ping` and `pong` *events* used previously become *channels* that carry and return state information to and from the buddy process – see Figure 7. Again, the proof of the equivalence between figures 6 and 7 (with the `ping`/`pong` channels hidden) follows directly from basic CSP algebra.

3.1.3 Applying Parallel Introduction

We can now deduce that the original system in Figure 2 is equivalent to the one in Figure 8.

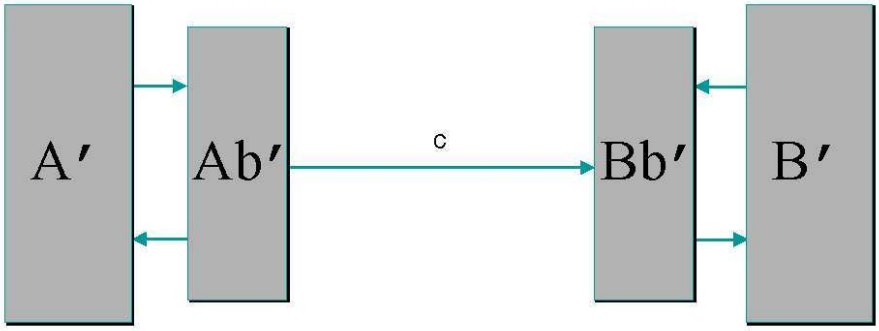

Figure 8: Parallel Introduction applied to Figure 2

`A′` and its buddy `Ab′` are derived from `A` as prescribed for *Parallel Introduction* – the same for `B′` and `Bb′` (derived from `B`). Notice that the buddy processes, `Ab′` and `Bb′`, are this time *completely specified*. They each have a simple loop in which they do their respective `ping`, their common channel communication, and then their respective `pong`s.

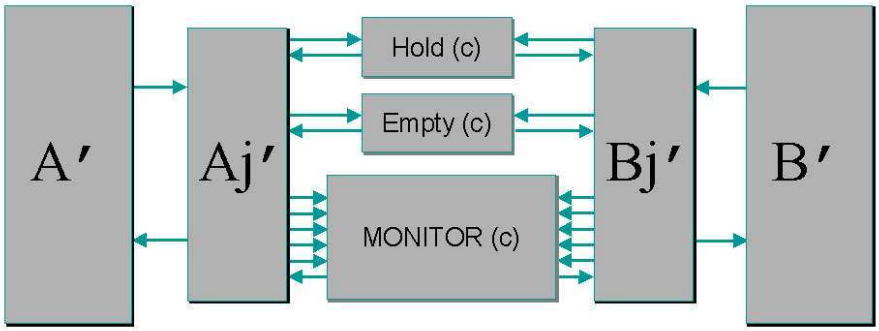

Figure 9: Parallel Introduction applied to Figure 3

We may also deduce, using *Parallel Introduction*, that the *Java-ised* system of Figure 3 is equivalent to that shown in Figure 9. Note that `Aj` has become the parallel composition of *the same* `A′` process as in Figure 8, but with a different buddy process `Aj′`. Processes `Ab′` and `Aj′` are identical except that the former does a CSP channel write and the latter does a JCSP one. Similarly, `Bj` from Figure 3 has become `B′` (*the same* as in Figure 8) in parallel with `Bj′` – and `Bj′` is related to `Bb′` as `Aj′` is to `Ab′`.

3.1.4 Applying the Model Checker

Looking at figures 8 and 9, we see that all processes, apart from `A′` and `B′`, are completely specified. If we can prove that the completely specified middles – i.e. figures 10 and 11 – are equivalent, we are done. This is because simple CSP laws of parallel composition (associativity) would allow us to deduce that figures 8 and 9 are equivelent and, hence, so must be our original systems in figures 2 and 3.

To prove figures 10 and 11 equivalent, we stand on the shoulders of giants and apply the FDR model checker. Both systems are fully specified and have the same channel interface (two pairs of `ping`/`pong` channels) to their environments – everything else is hidden. Type these systems into FDR and ask if they are equivalent. Within seconds, **Q.E.D!**

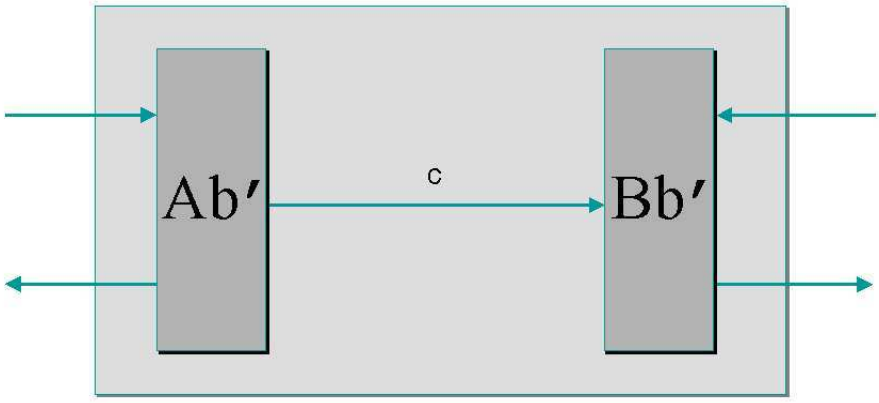

Figure 10: Completely Specified System – *model check* for equivalence with Figure 11

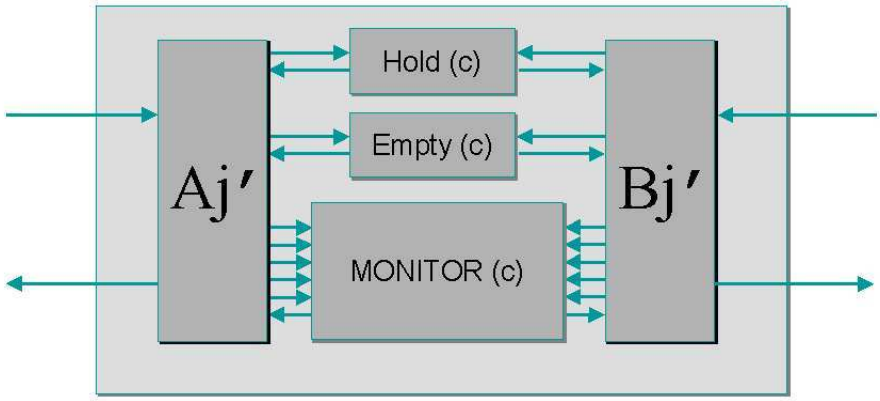

Figure 11: Completely Specified System – *model check* for equivalence with Figure 10

3.1.5 Connecting this Visualisation with the Rest of this Paper

The above rationalisation was constructed *after* the detailed proof reported in Sections 3.2, 3.3, 3.4 and 4 was developed.

Section 3.2 develops a CSP model of the JCSP channel that already includes *Parallel Introduction*. The *READ(o,t)* and *WRITE(o,t)* processes correspond to the buddy processes `Aj′` and `Bj′` from Figure 9. The *write.o.t* and *ack.o.t* channels are `ping` and `pong` for `Aj′`. The *ready.o.t* and *read.o.t* are `ping` and `pong` channels for `Bj′`.

In Section 3.3, the *LEFT* and *RIGHT* correspond to `Ab′` and `Bb′` from Figure 10. Their respective `ping` and `pong` channels are, of course, the same as those for `Aj′` and `Bj′`.

Finally, Section 4 performs the *Parallel Introduction* lemma in reverse (i.e. *Parallel Removal*) to go from Frigure 8 to Figure 2.

3.2 The CSP Model of the JCSP Channel

The JCSP channel, `One2OneChannel`, is currently implemented as a (100% pure) Java monitor. The following CSP model is derived directly from its source code and the CSP model of Java monitors just presented. This derivation was done by hand, but a tool could be built to assist this process considerably. Here is the outline of the class:

```
public class One2OneChannel {

  // data in transit
  private Object channel_hold;

  // synchronisation flag
  private boolean channel_empty = true;

  ...  public sync Object read ()
  ...  public sync void write (Object mess)

}
```

A JCSP channel object has two attributes (*channel_empty* and *channel_hold*), which we shall model as processes always ready both to have their values reset or to report them to any willing thread. For simplicity, we assume that each channel carries the same three-valued type (*Data*) that we shall use for the Java 'boolean'.

datatype $Variables = channel\_empty \mid channel\_hold$
datatype $Data = TRUE \mid FALSE \mid OTHER$

The operation of the channel, however, is independent of the type of data that it carries – at no point is the value of the data it stores used to decide its future behaviour. So this analysis is equally valid for channels of any type[1].

Variables are initialised as *TRUE*. Their values may then be read or written by any thread using channels *getvar* and *setvar*.

channel $getvar, setvar : Objects.Variables.Threads.Data$

$$VARIABLE(o, v) = VAR2(o, v, TRUE)$$

$$VAR2(o, v, d) = \left(\Box_{t \in Threads} getvar.o.v.t!d \rightarrow VAR2(o, v, d)\right) \Box \left(\Box_{t \in Threads} setvar.o.v.t?x \rightarrow VAR2(o, v, x)\right)$$

$$\alpha VARIABLE(o, v) = \{getvar.o.v.t.d, setvar.o.v.t.d \mid t \in Threads, d \in Data\}$$

$$VARIABLES(o) = VARIABLE(o, channel\_empty) \parallel VARIABLE(o, channel\_hold)$$

One purpose of JCSP is to seal off the thread/monitor synchronisation calls from the programmer. Instead a read/write interface is provided by two simple methods. We shall model this interface with the following events:

- $write.o.t.d$ – thread t invokes java method $write(d)$ of object o, message d is supplied for transmission;
- $ack.o.t$ – call to $write(d)$ terminates;
- $ready.o.t'$ – thread t' invokes method $read()$ of object o;
- $read.o.t'.d$ – call to $read()$ terminates, returning d.

[1] A formal theory of data independence in CSP has been developed by Ranko Lazic and Bill Roscoe[13].

The JCSP channel should behave like a synchronised channel. Each successful communication requires that at some point both threads are simultaneously involved.

channel *read*, *write* : *Objects*.*Threads*.*Data*
channel *ready*, *ack* : *Objects*.*Threads*

The Java code for the JCSP read method is as follows[14]:

```
public synchronized Object read ()
 throws InterruptedException {
  if (channel_empty) {
    channel_empty = false;       // first to the rendezvous
    wait ();                     // wait for writer process
    notify ();                   // schedule the writer to finish
  } else {
    channel_empty = true;        // second to the rendezvous
    notify ();                   // schedule the waiting writer
  }
  return channel_hold;
}
```

We model this JCSP read method as a process which repeatly waits to be activated by a ready event and then executes the monitor synchronisation code to receive a message:

$$
\begin{array}{l}
READ(o,t) = \\
\quad ready.o.t \rightarrow \\
\quad claim.o.t \rightarrow \\
\quad getvar.o.channel\_empty.t?c \rightarrow \\
\quad \left(\begin{array}{l} \textbf{if } (c = TRUE) \textbf{ then} \\ \quad setvar.o.channel\_empty.t!FALSE \rightarrow \\ \quad WAIT(o,t); \\ \quad NOTIFY(o,t) \\ \quad \textbf{else} \\ \quad setvar.o.channel\_empty.t!TRUE \rightarrow \\ \quad NOTIFY(o,t) \end{array} \right); \\
\quad getvar.o.channel\_hold.t?mess \rightarrow \\
\quad release.o.t \rightarrow \\
\quad read.o.t!mess \rightarrow \\
\quad READ(o,t)
\end{array}
$$

$$
\begin{array}{l}
\alpha READ(o,t) = \\
\quad \left\{ \begin{array}{l} claim.o.t, getvar.o.v.t.d, notify.o.t, \\ notifyall.o.t, setvar.o.v.t.d, read.o.t.d, \\ release.o.t, waita.o.t, waitb.o.t, \\ ready.o.t \mid v \in Variables, d \in Data \end{array} \right\}
\end{array}
$$

By including *all* the relevant java synchronisation events, such as *notifyall.o.t* in the alphabet of the CSP *READ* process we model our intention to prohibit the user of JCSP from calling the corresponding java methods directly from within his or her code. JCSP is intended as a complete, user-friendly alternative to using monitors for programming multi-threaded applications.

The Java code for the JCSP write method is as follows:

```
public synchronized void write (Object mess)
 throws InterruptedException {
  channel_hold = mess;
  if (channel_empty) {
    channel_empty = false;      // first to the rendezvous
    wait ();                    // wait for reader process
  } else {
    channel_empty = true;       // second to the rendezvous
    notify ();                  // schedule the waiting reader
    wait();                     // let the reader regain the lock
  }
}
```

This write method is similarly modelled as a repeating process, activated by the write event:

$$
\begin{array}{l}
WRITE(o,t) = \\
\quad write.o.t?mess \rightarrow \\
\quad claim.o.t \rightarrow \\
\quad setvar.o.channel\_hold.t!mess \rightarrow \\
\quad getvar.o.channel\_empty.t?c \rightarrow \\
\quad \left(\begin{array}{l} \textbf{if } (c = TRUE) \textbf{ then} \\ \quad setvar.o.channel\_empty.t!FALSE \rightarrow \\ \quad WAIT(o,t) \\ \textbf{else} \\ \quad setvar.o.channel\_empty.t!TRUE \rightarrow \\ \quad NOTIFY(o,t); \\ \quad WAIT(o,t) \end{array} \right); \\
\quad release.o.t \rightarrow \\
\quad ack.o.t \rightarrow \\
\quad WRITE(o,t)
\end{array}
$$

$$
\alpha WRITE(o,t) = \left\{ \begin{array}{l} claim.o.t, getvar.o.v.t.d, notify.o.t, \\ notifyall.o.t, setvar.o.v.t.d, write.o.t.d, \\ release.o.t, waita.o.t, waitb.o.t, \\ ack.o.t \mid v \in Variables, d \in Data \end{array} \right\}
$$

The JCSP channel is then modelled as the parallel composition of processes which model its monitor, two attributes, and read and write methods:

$$
\begin{array}{rcl}
JCSPCHANNEL(o,t_1,t_2) & = & READ(o,t_1) \parallel \\
 & & WRITE(o,t_2) \parallel \\
 & & MONITOR(o) \parallel \\
 & & VARIABLES(o)
\end{array}
$$

Note that there is an implicit assumption here that the *read* and *write* methods of the JCSP channel will only be used by those threads for which they are intended. Otherwise we would have to include a separate parallel *READ* and *WRITE* process for each thread t in *Threads*.

Note also that in the above definition (and in the rest of this paper) we shall assume that the alphabet of a parallel composition is the union of the alphabets of the component processes (in line with Hoare's book[5]).

3.3 Equivalence to a Simpler Channel

Now we shall define a simplified model of how the JCSP channel should work, and then use FDR to show that this is equivalent to the JCSP implementation.

The simple channel consists of two parallel processes, *LEFT* and *RIGHT*, to handle input and output respectively. The processes are joined by a hidden channel *transmit* – see Figure 12.

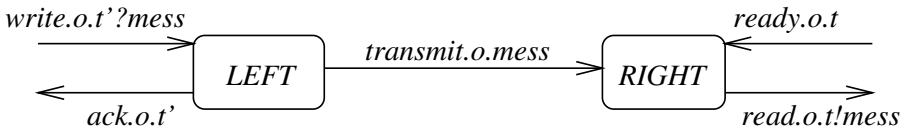

Figure 12: Special Version of Figure 10

We define:

channel *transmit* : *Objects*.*Data*

$$LEFT(o,t) = \\ \quad write.o.t?mess \rightarrow transmit.o!mess \rightarrow ack.o.t \rightarrow LEFT(o,t)$$

$$\alpha LEFT(o,t) = \\ \quad \{write.o.t.m, transmit.o.m, ack.o.t \mid m \in Data\}$$

$$RIGHT(o,t') = \\ \quad ready.o.t' \rightarrow transmit.o?mess \rightarrow read.o.t'!mess \rightarrow RIGHT(o,t')$$

$$\alpha RIGHT(o,t') = \\ \quad \{ready.o.t', transmit.o.m, read.o.t'.m \mid m \in Data\}$$

$$CHANNEL(o,t,t') = \\ \quad (LEFT(o,t') \parallel RIGHT(o,t)) \setminus \{transmit.o.m \mid m \in Data\}$$

$$\alpha CHANNEL(o,t,t') = \\ \quad \{\ write.o.t'.m, ack.o.t', ready.o.t, read.o.t.m \mid m \in Data\ \}$$

In order to compare this specification with the JCSP implementation we need to conceal all the additional events in the alphabet of *JCSPCHANNEL*.

$$Private \quad = \quad \alpha JCSPCHANNEL(0,0,1) - \alpha CHANNEL(0,0,1)$$

The CSP language of Hoare is a notation for describing patterns of communication by algebraic expressions. It is widely used for the design of parallel and distributed hardware and software, and for the formal proof of vital properties of such systems. Underpinning CSP is a formal semantic model based on *traces*, *failures* and *divergences*. Two CSP systems are equivalent if the possible sequences of events (traces) they may perform are identical, and also if the circumstances under which they might deadlock or livelock are the same.

We assert that $JCSPCHANNEL(o, t_1, t_2) \setminus Private$ is equivalent to $CHANNEL(o, t_1, t_2)$ in the failures/divergences model:

$$\textbf{assert}\ CHANNEL(0,0,1) = JCSPCHANNEL(0,0,1) \setminus Private$$

The FDR[2] tool can check for equivalence between CSP systems. The sizes of the problems it may tackle are limitied to around one billion states with current workstation technologies. The above assertion is verified within seconds using this tool.

3.4 Interference by Other Threads

The above works fine with the current system when only two threads are in existence. When we increase the number of threads in the system beyond two, perhaps by defining $Threads = \{0,1,2\}$, we find that the above pair of assertions no longer holds. FDR reveals that this is because the other threads may tamper with the state of the channel implementation, using the *getvar* and *setvar* channels.

So how do we stop other threads from interfering with the channel object? In CSP we can add a parallel process to the channel implementation which blocks access to any of the relevant events!

$$PROTECTION(o,t,t') = STOP$$

$$\alpha PROTECTION(o,t,t') = \left\{ \begin{array}{l} claim.o.t'', setvar.o.v.t''.d, \\ getvar.o.v.t''.d, waita.o.t'' \\ \mid t'' \in Threads - \{t,t'\}, \\ v \in Variables, \\ d \in Data \end{array} \right\}$$

$$SAFEJCSPCHANNEL(o,t,t') = JCSPCHANNEL(o,t,t') \parallel PROTECTION(o,t,t')$$

$$\textbf{assert}\ CHANNEL(0,0,1) = SAFEJCSPCHANNEL(0,0,1) \setminus Private$$

This pair of assertions is indeed found to hold when the number of threads is increased beyond 2.

This is not supported by the actual Java implementation of `One2OneChannel` and so must be regarded as a usage rule for JCSP. The same usage rule is, of course, enforced for occam channels by its semantics (and by its compilers). Given that JCSP channels are just Java objects (i.e. held by reference to stack addresses), that usage rule must be enforced either manually or by good design tools.

However, JCSP also provides `Any2OneChannels` (as well as `One2AnyChannel` and `Any2AnyChannel`) that do give the necessary protection to control parallel reads and writes. This is considered in Section 5. First, we must complete the proof of correctness for `One2OneChannel`.

4 Correctness of the JCSP Channel

So far we have succeeded in reducing the JCSP channel to a vastly simplified form, which does away with monitors, and involves just two very simple processes: *LEFT* and *RIGHT*. This is useful but it would be nice to go one stage further and be rid of these two processes, leaving just a single unprotected CSP channel. It turns out that this is only possible if we

assume certain 'usage' rules about how networks are constructed using JCSP, similar to those enforced by occam compilers in that alternative implementation of CSP.

For the moment let us consider simple 'ALT-free' CSP programs that use only one-to-one channels and no have alternation. Define an SCSP network as a special kind of parallel CSP network $P_1 \parallel P_2 \parallel .. \parallel P_n$, where each P_i is an SCSPPROC:

$$\begin{aligned} SCSPPROC &= SKIP \\ &\mid a!x \rightarrow SCSPPROC \\ &\mid a?x \rightarrow SCSPPROC(x) \\ &\mid SCSPPROC \sqcap SCSPPROC \\ &\mid SCSPPROC;\ SCSPPROC \end{aligned}$$

A usage rule is enforced which is that each channel a is used by exactly one process P_i for input and exactly one process P_j for output, i.e. the network is triple-disjoint. (There will be other external events though to represent things like reading in data and printing out results).

So we have described a simple set of CSP processes that we would like to model using JCSP. However there is an obstacle to this – we don't really have any CSP channels available for use - we only have JCSP channels, which behave like extra parallel processes:

$$\begin{aligned} JCSPCHANNEL(a) &= (LEFT(a) \parallel RIGHT(a)) \setminus \{a\} \\ \text{where } LEFT(a) &= write.a?x \rightarrow a!x \rightarrow ack.a \rightarrow LEFT(a) \\ \text{and } RIGHT(a) &= ready.a \rightarrow a?x \rightarrow read.a!x \rightarrow RIGHT(a) \end{aligned}$$

This representation of a JCSP channel has been proven correct above using FDR.

We are now going to show that these JCSP channels can be used just like ordinary CSP channels. We shall consider an SCSP network $V = P_1 \parallel .. \parallel P_n$ and transform it in some way that replaces all the CSP channels with JCSP channel processes, and show that the external behaviour of the program is preserved.

Let us define the transformation as follows: suppose that network V originally contains a CSP channel a, which is written to by process P_i and read from by process P_j. To replace CSP channel a in V we introduce an additional parallel process $JCSPCHANNEL(a)$, and we transform process P_i to P_i' by replacing all occurrences of $a!x \rightarrow Process$ by $write.a!x \rightarrow ack.a \rightarrow Process$. And we transform process P_j to P_j' by replacing all occurrences of $a?x \rightarrow Process(x)$ by $ready.a \rightarrow read.a?x \rightarrow Process(x)$

Now, because of the nice algebraic laws of CSP, if we can show that the external behaviour of subnetwork $P_i \parallel P_j$ is unchanged by this transformation then it follows that there is no effect on the external behaviour of the network as a whole.

What we actually need to prove is that:

$$(P_i \parallel P_j) \setminus \{a\} \quad = \quad (P_i' \parallel JCSPCHANNEL(a) \parallel P_j') \setminus \{write.a, ack.a, ready.a, read.a\} \tag{1}$$

Let's start with the RHS:

$$(P_i' \parallel JCSPCHANNEL(a) \parallel P_j') \setminus \{write.a, ack.a, ready.a, read.a\} =$$

$$(P_i' \parallel ((LEFT(a) \parallel RIGHT(a)) \setminus a) \parallel P_j') \setminus \{write.a, ack.a, ready.a, read.a\} =$$

$$(P_i' \parallel LEFT(a) \parallel RIGHT(a) \parallel P_j') \setminus \{a, write.a, ack.a, ready.a, read.a\} =$$

$$\left(\begin{array}{c} ((P_i' \parallel LEFT(a)) \setminus \{write.a, ack.a\}) \parallel \\ ((RIGHT(a) \parallel P_j') \setminus \{ready.a, read.a\}) \end{array} \right) \setminus \{a\}$$

We can establish result (1) if we can prove the following:

$$(P_i' \parallel LEFT(a)) \setminus \{write.a, ack.a\} = P_i \quad (2)$$

$$\text{and } (RIGHT(a) \parallel P_j') \setminus \{ready.a, read.a\} = P_j \quad (3)$$

Let us consider the validity of equation 2. This would not hold true in general for any CSP process P_i. But due to our restriction on the syntax of SCSP processes, we can see that it is true as follows.

1. In moving from P_i to P_i' we replace $a!x \rightarrow PROCESS$ with $write.a!x \rightarrow ack.a \rightarrow PROCESS$
2. The effect of putting $LEFT(a)$ in parallel with P_i' is then to replace $write.a!x \rightarrow ack.a \rightarrow PROCESS$ in P_i' with $write.a!x \rightarrow a!x \rightarrow ack.a \rightarrow PROCESS$
3. The effect of hiding $\{write.a, ack.a\}$ is to set $write.a!x \rightarrow a!x \rightarrow ack.a \rightarrow PROCESS$ back to $a!x \rightarrow PROCESS$, which is what we started with. This is the only part of the proof which makes use of the restricted syntax. For if we were to allow external choice to be applied to JCSP channels, then hiding the *write* channel would introduce unwanted non-determinism.

Equation 3 is proved similarly and we conclude that the transformation to replace CSP channel a with $JCSPCHANNEL(a)$ has not affected the external behaviour of the network.

Having applied a transformation to replace one CSP channel a' with a JCSP channel we can repeat the step on other channels until only JCSP channels remain. (This is because the transformation from P_i and P_j to P_i' and P_j' preserves the restricted syntax of the SCSP processes). Finally we will have shown that

$$(P_1 \parallel .. \parallel P_n) \setminus \{a_1, .., a_m\} = \left(\begin{array}{c} (P_1' \parallel .. \parallel P_n') \parallel \\ \left(\begin{array}{c} JCSPCHANNEL(a_1) \parallel \\ .. \parallel JCSPCHANNEL(a_m) \end{array} \right) \end{array} \right) \setminus \left\{ \begin{array}{l} write.a_1, ack.a_1, ready.a_1, read.a_1, .. \\ write.a_m, ack.a_m, ready.a_m, read.a_m \end{array} \right\}$$

Which means that the external behaviour of the network (i.e its behaviour when all internal channels are concealed) is exactly preserved.

So we are perfectly safe to reason about JCSP programs in their natural form, modelling calls to *read* and *write* as atomic communication events. There is no need for the additional baggage of *LEFT*/*RIGHT* process pairs for each channel.

5 Any-to-One *(Shared)* Channels

A serious omission from the JCSP model considered so far is (occam) *Alternation.* We consider here a simplified version that affords a much more efficient implementation. It corresponds to the occam3 notion of a *SHARED* channel. In JCSP, this is the `Any2One-Channel`, which allows any number of concurrent writers but only a single reader.

`Any2OneChannel` is similar to `One2OneChannel`, but contains an extra attribute – `write_monitor`. This is a dumb object whose only use is to provide a monitor lock that must be acquired by a writer before writing. The `read` method is unchanged from `One2OneChannel`. The modified `write` method is as follows:

```
public void write (Object mess)
 throws InterruptedException {
  synchronized (write_monitor) {      // compete with other writers
    synchronized (this) {             // compete with a single reader
      channel_hold = mess;
      if (channel_empty) {
        channel_empty = false;        // first to the rendezvous
        wait ();                      // wait for reader process
      } else {
        channel_empty = true;         // second to the rendezvous
        notify ();                    // schedule the waiting reader
        wait();                       // let the reader regain the lock
      }
    }
  }
}
```

For the CSP version, an extra object is needed for the write monitor.

$$
\begin{array}{l}
WRITE'(o, writemonitor, t) = \\
\quad write.o.t?mess \rightarrow \\
\quad claim.writemonitor.t \rightarrow \\
\quad claim.o.t \rightarrow \\
\quad setvar.o.channel\_hold.t!mess \rightarrow \\
\quad getvar.o.channel\_empty.t?c \rightarrow \\
\quad \left(\begin{array}{l}
\quad \textbf{if } (c = TRUE) \textbf{ then} \\
\quad setvar.o.channel\_empty.t!FALSE \rightarrow \\
\quad WAIT(o, t) \\
\quad \textbf{else} \\
\quad setvar.o.channel\_empty.t!TRUE \rightarrow \\
\quad NOTIFY(o, t); \\
\quad WAIT(o, t)
\end{array} \right); \\
\quad release.o.t \rightarrow \\
\quad release.writemonitor.t \rightarrow \\
\quad ack.o.t \rightarrow \\
\quad WRITE'(o, writemonitor, t)
\end{array}
$$

$$
\begin{array}{l}
\alpha WRITE'(o, o', t) = \\
\quad \left\{ \begin{array}{l}
claim.o''.t, getvar.o.v.t.d, notify.o''.t, \\
notifyall.o''.t, setvar.o.v.t.d, write.o.t.d, \\
release.o''.t, waita.o''.t, waitb.o''.t, ack.o.t \\
\mid v \in Variables, d \in Data, o'' \in \{o, o'\}
\end{array} \right\}
\end{array}
$$

The read method is essentially the same as before, except that we need to include certain events in its alphabet to prevent the reading thread from interfering with the write monitor. (This was only discovered after FDR caught a livelock on an early run.)

$$READ'(o, o', t) = READ(o, t)$$

$$\alpha READ'(o, o', t) = \alpha READ(o, t) \cup \{claim.o'.t\}$$

$$
\begin{array}{l}
JCSPSHAREDCHANNEL(o, o', t_1, t_2, t_3) = \\
\quad READ'(o, o', t_1) \parallel \\
\quad WRITE'(o, o', t_2) \parallel WRITE'(o, o', t_3) \parallel \\
\quad MONITOR(o) \parallel MONITOR(o') \parallel VARIABLES(o)
\end{array}
$$

Now we arrive at the step of reducing the CSP representation of an `Any2OneChannel` to a simpler equivalent form. Since the FDR tool can perform only static analysis of finite systems we shall restrict ourselves to the case of a two-to-one channel. (However it should be possible to extend this analysis to the general case by a form of mathematical induction devised by Creese and Roscoe for CSP programs with arbitrary network topologies[1].)

The specification for a two-to-one JCSP channel consists of three simple processes – one for each user thread (see Figure 13):

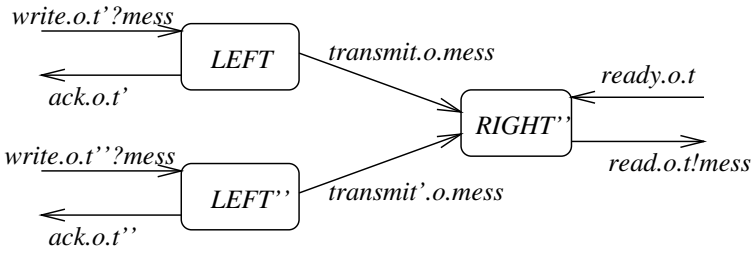

Figure 13: A 2-way Shared Channel Version of Figure 10

We define:

channel $transmit' : Objects.Data$

$$LEFT''(o,t) = write.o.t?mess \rightarrow transmit'.o!mess \rightarrow ack.o.t \rightarrow LEFT''(o,t)$$

$$\alpha LEFT''(o,t) = \left\{ \begin{array}{l} write.o.t.m, transmit'.o.m, \\ ack.o.t \mid m \in Data \end{array} \right\}$$

$$RIGHT''(o,t') = ready.o.t' \rightarrow \left(\begin{array}{l} transmit.o?mess \rightarrow read.o.t'!mess \rightarrow RIGHT''(o,t') \ \Box \\ transmit'.o?mess \rightarrow read.o.t'!mess \rightarrow RIGHT''(o,t') \end{array} \right)$$

$$\alpha RIGHT''(o,t') = \left\{ \begin{array}{l} ready.o.t', transmit.o.m, transmit'.o.m, \\ read.o.t'.m \mid m \in Data \end{array} \right\}$$

$$SHAREDCHANNEL(o,t,t',t'') = \left(LEFT(o,t') \parallel LEFT''(o,t'') \parallel RIGHT''(o,t) \right) \setminus \{transmit.o.m, transmit'.o.m \mid m \in Data\}$$

In order to compare this specification with the JCSP implementation we need to conceal all the additional events in the alphabet of *JCSPSHAREDCHANNEL*.

$$Private2 = \alpha JCSPSHAREDCHANNEL(0,1,0,1,2) - \alpha SHAREDCHANNEL(0,0,1,2)$$

Assert that $JCSPSHAREDCHANNEL(o,o',t_1,t_2,t_3) \setminus Private2$ is equivalent to $SHAREDCHANNEL(o,t_1,t_2,t_3)$ in the Failures/Divergences model.

assert $SHAREDCHANNEL(0,0,1,2) = JCSPSHAREDCHANNEL(0,1,0,1,2) \setminus Private2$

Again, this is easily verified using FDR.

The analysis of Section 4 can now be extended to cater for simple networks with shared CSP channels.

6 Analysis of the CTJ Channel

Having proved correctness for the JCSP channel it is natural to move on to consideration of its rival: CTJ (*Communicating Threads for Java*)[7].

The CTJ channel algorithm is quite similar to JCSP, but it handles the monitor synchronisation in a subtly different, and somewhat simpler way. CTJ only has *any-to-any* channels. The following shows the core *one-to-one* code that it uses, written in the same style as the JCSP version presented in Section 3.2:

```
public class One2OneChannel {

  private Object channel_hold;           // data in transit (as in JCSP)

  private boolean channel_empty = true;  // sync flag (used differently)

  // the above flag indicates whether there has been a write.
  // previously, it indicated whether there had been a read or a write.

  public synchronized Object read () throws InterruptedException {
    if (channel_empty) {
      wait ();                           // wait for a writer
    }
    // there has been a write and channel_empty is now definitely false.
    channel_empty = true;
    notify ();                           // there is a writer waiting
    return channel_hold;                 // this will still be valid
  }

  public synchronized void write (Object value) throws InterruptedException {
    channel_empty = false;
    channel_hold = value;
    notify ();                           // redundant if first to the channel
    wait ();                             // wait for a reader
  }

}
```

This is implemented in our CSP model by redefining the *READ* and *WRITE* processes as follows:

$$
\begin{aligned}
&READ(o,t) = \\
&\quad ready.o.t \rightarrow claim.o.t \rightarrow \\
&\quad getvar.o.channel\_empty.t?c \rightarrow (\textbf{if}\,(c = TRUE)\,\textbf{then}\,WAIT(o,t)\textbf{else}\,SKIP)\,; \\
&\quad setvar.o.channel\_empty.t!TRUE \rightarrow NOTIFY(o,t); \\
&\quad getvar.o.channel\_hold.t?mess \rightarrow \\
&\quad release.o.t \rightarrow read.o.t!mess \rightarrow READ(o,t)
\end{aligned}
$$

$$
\begin{aligned}
&WRITE(o,t) = \\
&\quad write.o.t?mess \rightarrow claim.o.t \rightarrow \\
&\quad setvar.o.channel\_empty.t!FALSE \rightarrow \\
&\quad setvar.o.channel\_hold.t!mess \rightarrow \\
&\quad NOTIFY(o,t);\ WAIT(o,t); \\
&\quad release.o.t \rightarrow ack.o.t \rightarrow WRITE(o,t)
\end{aligned}
$$

The channel is then shown to be equivalent to the simpler form, and hence also the JCSP channel, in the same manner as before:

$$\textbf{assert}\ CHANNEL(0,0,1) = JCSPCHANNEL(0,0,1) \setminus Private$$

7 Verifying the JCSP ALT Construct

General *ALT*ing provides much more control (albeit at $O(n)$-cost) than the shared channels considered in Section 5. For example, shared channels do not enable a process to listen exclusively to channel x one moment and channels x and y another - so *ALTing* is crucial.

Further, it was the (published[15], compact and well-used) JCSP implementation of *ALTing* that contained the unfortunate race hazard that eventually yielded the deadlock mentioned in Section 1. This requires the analysis of many interacting monitors – the `Alternative` object itself (which has two methods) and the channels (which are extended to four methods).

We have recently performed a formal verification of the current JCSP `Alternative` class (given below). This was done in a fairly restricted manner: considering only two channels firing at a single alternation construct. However, its success has brought great relief and given us confidence in the correctness of the *n-way ALT* with *pre-conditions*.

The analysis was considerably more complex than that of the single and shared channels, and so we shall only include an outline of the methods employed here. The complete FDR document is available for reference at [11].

```
public class Alternative {

  private static final int enabling = 0;
  private static final int waiting = 1;
  private static final int ready = 2;
  private static final int inactive = 3;
  private int state = inactive;
  private final Channel[] c;

  public Alternative (final Channel[] c( {
    this.c = c;
  }

  public int select () throws InterruptedException {
    int selected = -999999;    // this value should *never* be returned!
    int i;
    state = enabling;                                  // ALT START
    for (i = 0; i < c.length; i++) {
      if (c[i].enable (this)) {                        // ENABLE CHANNEL
        state = ready;
        selected = i;
        break;
      }
    }
    synchronized (this) {
      if (state == enabling) {                         // ALT WAIT
        state = waiting;
        wait ();
        state = ready;
      }
    }
    // assert : state == ready
    for (i--; i >= 0; i--) {
      if (c[i].disable ()) {                           // DISABLE CHANNEL
        selected = i;
      }
    }
    state = inactive;
    return selected;                                   // ALT END
  }
```

```
  synchronized void schedule () {
    switch (state) {
      case enabling:
        state = ready;
      break;
      case waiting:
        state = ready;
        notify ();
      break;
      // case ready: case inactive:
      // break
    }
  }
}
```

Here is the modified `Channel` class, which allows for alternation:

```
public class Channel {

  private int channel_hold;             // buffer (not detectable to users)
  private boolean channel_empty = true;// synchronisation flag
  private Alternative alt;              // state of reader

  public synchronized int read () throws InterruptedException {
    if (channel_empty) {
      channel_empty = false;            // first to the rendezvous
      wait ();                          // wait for the writer thread
      notify ();                        // schedule the writer to finish
    } else {
      channel_empty = true;             // second to the rendezvous
      notify ();                        // schedule the waiting writer thread
    }
    return channel_hold;
  }

  public synchronized void write (int n) throws InterruptedException
    channel_hold = n;
    if (channel_empty) {
      channel_empty = false;            // first to the rendezvous
      if (alt != null) {                // the reader is ALTing on this Channel
        alt.schedule ();
      }
      wait ();                          // wait for the reader thread
    } else {
      channel_empty = true;             // second to the rendezvous
      notify ();                        // schedule the waiting reader thread
      wait ();                          // let the reader regain this monitor
    }
  }

  synchronized boolean enable (Alternative alt) {
    if (channel_empty) {
      this.alt = alt;
      return false;
    } else {
      return true;
    }
  }
```

```
  synchronized boolean disable () {
    alt = null;
    return !channel_empty;
  }
}
```

For the purpose of verification we modelled a system of two channels interacting with a single alternation construct. Even this resulted in a fairly complicated CSP network as shown in Figure 14.

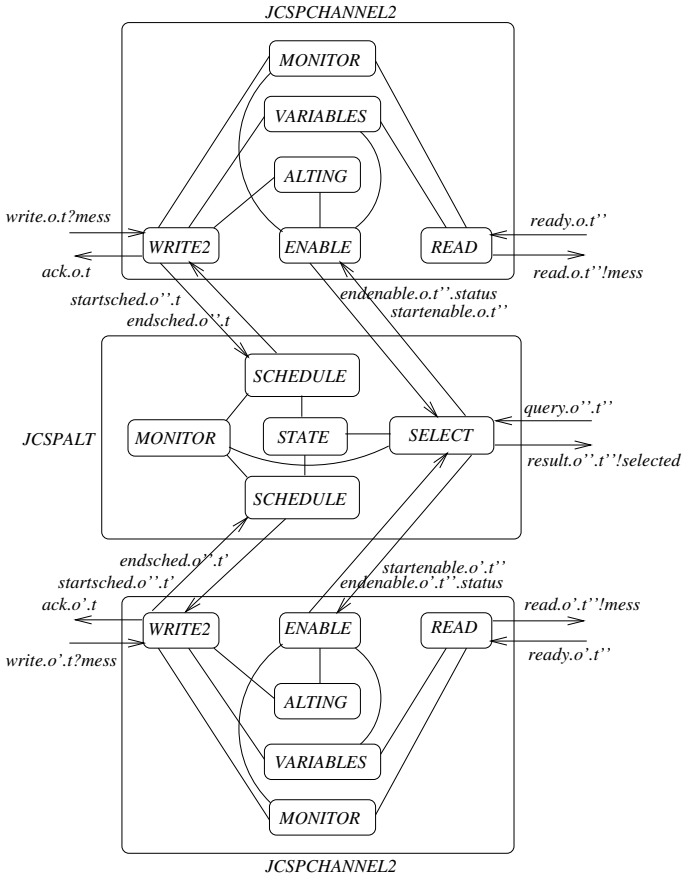

Figure 14: A 2-way ALTing Channel Version of Figure 11

The code for the various CSP processes is too long to be included here, but it is a straightforward translation of the equivalent Java (in the same way as for the `One2OneChannel`. The function of those processes not defined above is as follows.

- *WRITE2* – `Channel.write` method, modified to keep track of `alt` variable and to invoke `Alternative.schedule`.
- *ENABLE* – models both `Channel.enable` and `Channel.disable`.
- *ALTING* – process to model the `alt` variable.
- *SCHEDULE* – `Alternative.schedule` method (there are two copies of this: one to service each channel).
- *STATE* – process to model the state variables for `Alternate`: `state, selected` and `i`.
- *SELECT* – `Alternative.select` method.

For our specification, which we wish to prove equivalent to the alternation construction, we consider two simple *CHANNEL* processes placed in parallel with an *ALT* process which snoops their *write* channels in order to provide the reader with selection information:

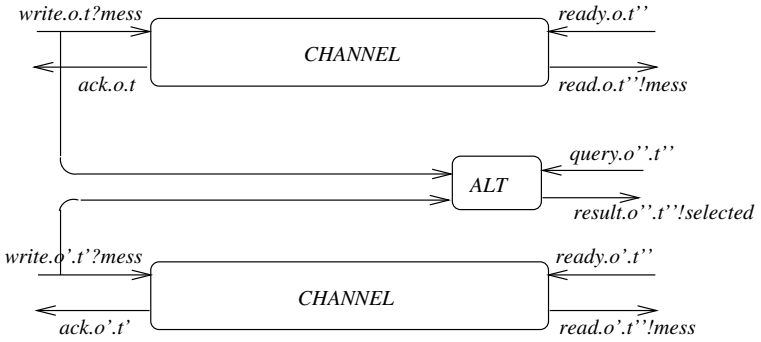

Figure 15: A 2-way ALTing Channel Version of Figure 10

The CSP code for the *ALT* process is as follows.

$$
\begin{array}{l}
ALT(o'',t'',o,t,o',t',ready0,ready1,waiting) = \\
\quad write.o.t?mess \rightarrow ALT(o'',t'',o,t,o',t',TRUE,ready1,waiting) \\
\quad \Box \\
\quad write.o'.t'?mess \rightarrow ALT(o'',t'',o,t,o',t',ready0,TRUE,waiting) \\
\quad \Box \\
\left(\begin{array}{l}
\quad \textbf{if}\ (waiting = TRUE)\ \textbf{then}\ (\\
\qquad \textbf{if}\ (ready0 = TRUE\ \textbf{and}\ ready1 = TRUE)\ \textbf{then}\ (\\
\qquad\quad (return.o''.t''!0 \rightarrow ALT(o'',t'',o,t,o',t',FALSE,ready1,FALSE)) \\
\qquad\quad \Box \\
\qquad\quad (return.o''.t''!1 \rightarrow ALT(o'',t'',o,t,o',t',ready0,FALSE,FALSE)) \\
\qquad)\textbf{else if}\ (ready1 = TRUE)\ \textbf{then}\ (\\
\qquad\quad return.o''.t''!1 \rightarrow ALT(o'',t'',o,t,o',t',ready0,FALSE,FALSE) \\
\qquad)\textbf{else if}\ (ready0 = TRUE)\ \textbf{then}\ (\\
\qquad\quad return.o''.t''!0 \rightarrow ALT(o'',t'',o,t,o',t',FALSE,ready1,FALSE) \\
\qquad)\textbf{else}\ STOP \\
\quad)\textbf{else}\ (\\
\qquad query.o''.t'' \rightarrow ALT(o'',t'',o,t,o',t',ready0,ready1,TRUE) \\
\quad)
\end{array} \right)
\end{array}
$$

In each state *ALT* maintains three variables: *ready0, ready1* and *waiting*. The two former variables record which channels are carrying data, and the latter records whether an unanswered query has been issued by the reader. Note that this is not a prioritized alternation construct. If both channels are carrying data then a non-deterministic choice is made between them.

Now in order to compare the two processes we have to add a final parallel component to each one to reflect a JCSP usage rule, which is that the reader must first call the `select` method and then call which ever `read` method is appropriate. We are not interested in comparing how the specification and implementation would behave under an illegal usage pattern.

$$
\begin{array}{l}
USAGE(o'',t'',o,o') = \\
\quad query.o''.t'' \rightarrow return.o''.t''?c \rightarrow \\
\quad \textbf{if}\ (c = 0)\ \textbf{then}\ (\\
\qquad ready.o.t'' \rightarrow read.o.t''?mess \rightarrow USAGE(o'',t'',o,o') \\
\quad)\textbf{else}\ (\\
\qquad ready.o'.t'' \rightarrow read.o'.t''?mess \rightarrow USAGE(o'',t'',o,o') \\
\quad)
\end{array}
$$

FDR, after working through around 48,000 states, confirms that the implementation is equivalent to the specification – which is a good step towards total confidence in the JCSP ALT code. However there is still further work that could be done to consider alternation structures with pre-conditions, and those which contain more than two channels.

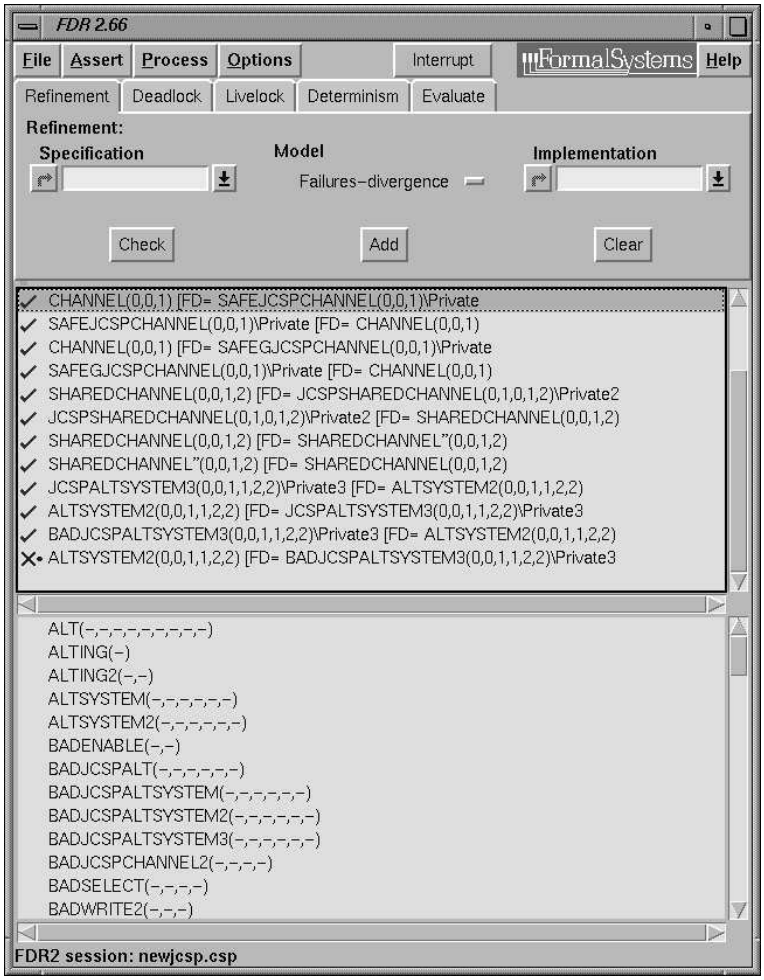

Figure 16: Snapshot of FDR Finding the Deadlock in the Original (Bad) JCSP `ALT`

The original faulty JCSP implementation was also analsyed using FDR and, sure enough, the deadlock trace that had taken two years to discover and then eliminate was revealed in a matter of seconds – Figure 16.

8 Conclusions and Future Work

The CSP model of Java's monitor primitives means that *any* multithreaded Java code – not just code using the JCSP or CTJ[7] libraries that give direct access to CSP primitives – becomes amenable to formal and (partly) automated analysis. This should be of interest to the Java community.

Earlier work in the area of model-checking concurrent Java programs has been performed by Naumovich *et al*[12]. Their approach involves specifying the behaviour of Java monitors as a collection of constraints over event orderings. These constraints are defined using finite transition systems. There is a certain similarity to the language of **sat** which is used to specify behavioural characteristics of CSP programs using set theory and logic[10]. However, theirs is a *traces-only* model and so provides no easy way to check for deadlocks. By using CSP as

our underlying model, we are able to test for a wide ranging set of properties: e.g. deadlock-freedom, livelock-freedom, program equivalence and program refinement. And we can use the powerful and mature FDR tool.

In general, Java designers, implementors, testers and maintainers are running scared of its multithreading primitives. However, applications force their use very quickly (e.g. to maintain decent response times from continuous systems with external control). The Mars *Pathfinder* mission of 1998 suffered a race hazard from just three interacting threads that led to real-time failure on the Martian surface [8]. In future, if not already, finance-critical and safety-critical applications will be multithreaded. How sure will the authors be of their security?

A CSP model of multithreading puts this on a solid engineering foundation. Noone can ignore that. Anyone doing so would, at the very least, be exposed to a risk of litigation after a messy accident gets traced back to a system failure arising from some race condition:

> "So, Bill, you sold your system without really knowing whether it was deadlock-free ... and you never even tried these standard procedures that might have found out?!! Disclaimer notices notwithstanding, would you say that that shows a lack of due care or a lack of diligence towards your customer?"

References

[1] S. J. Creese and A. W. Roscoe, *Formal Verification of Arbitrary Network Topologies*, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), Volume II. CSREA Press 1999

[2] Formal Systems (Europe) Ltd, *Failures-Divergence Refinement: FDR2 Manual*, 1997.

[3] Inmos Ltd, **occam3** *user manual*, 1991, `<http://www.hensa.ac.uk/parallel/occam/documentation/>`.

[4] H. Muller and K. Walrath, *Threads and Swing*, April 1998, `<http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html>`.

[5] C.A.R. Hoare, *Communicating Sequential Processes*, *Prentice-Hall*, 1985.

[6] C.A.R. Hoare, *The Emperor's Old Clothes*, 1980 Turing Award Lecture, Commun. ACM 30, 8, pp 672-686, February 1981.

[7] G. Hilderink, J. Broenink, W. Vervoort and A. Bakkers, *Communicating Java Threads*, in Parallel Programming and Java, WoTUG-20, pp. 48-76, IOS Press (Amsterdam), ISBN 90 5199 336 6, April 1997.

[8] N. Hess, *Pathfinder Debugging*, `<java.sun.com/people/jag/pathfinder.html>`.

[9] J.M.R. Martin and S.A. Jassim, *A Tool for Proving Deadlock Freedom*, in Parallel Programming and Java, proceedings of WoTUG-20, pp. 1-16, IOS Press (Amsterdam), ISBN 90 5199 336 6, April 1997.

[10] J.M.R. Martin, *A Tool for Checking the CSP* **sat** *Property*, The Computer Journal, Vol. 43, No. 1, 2000.

[11] J.M.R. Martin, *FDR analysis of Peter Welch's CSP model of Java threads and the JCSP implementation of a channel*, to be made available from the JCSP web site.

[12] G. Naumovich, G.S. Avrunin and L.A. Clarke, *Data Flow Analysis for Checking Properties of Concurrent Java Programs*, Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002), ISBN 1 58113 074 0, May 1999.

[13] A.W. Roscoe, *The Theory and Practice of Concurrency*, *Prentice-Hall*, 1998.

[14] P.H.Welch, *Java Threads in the Light of occam/CSP*, in Architectures, Languages and Patterns for Parallel and Distributed Applications, proceedings of WoTUG-21, pp. 259-284, IOS Press (Amsterdam), ISBN 90 5199 391 9, April 1998.

[15] P.H. Welch, *ALTing*, Java Threads Workshop (Post Workshop Discussion), Oct. 1996, <http://www.hensa.ac.uk/parallel/groups/wotug/java/discussion/4.html>.

[16] P.H. Welch, *A CSP model for Java Multithreading*, java-threads@ukc.ac.uk mailing list, March 1999, <http://www.cs.ukc.ac.uk/projects/ofa/java-threads/203.html>.

[17] P.H. Welch, *A CSP model for Java Multithreading*, P.H.Welch and J.M.R. Martin, in Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems, pp. 114-122, IEEE Cat. No. PR00634, ISBN 0 7695 0634 8, June 2000.

[18] P.H. Welch and P.D. Austin, *JCSP home page*, since 1999, <http://www.cs.ukc.ac.uk/projects/ofa/jcsp/>.