A CSP Model for Java Threads (and Vice-Versa)

Peter Welch
Computing Laboratory
University of Kent
(`P.H.Welch@ukc.ac.uk`)

Jeremy M. R. Martin
Oxagen Limited
Abingdon, Oxfordshire
(`j.martin@oxagen.co.uk`)

Logic and Semantics Seminar (CU Computer Laboratory)
(16th. March, 2000)

The Real World and Concurrency

Computer systems - to be of use in this world - need to model that part of the world for which it is to be used.

If that modeling can reflect the natural concurrency in the system … it should be **simpler**.

Yet concurrency is thought to be an **advanced** topic, **harder** than serial computing (which therefore needs to be mastered first).

This tradition is WRONG!

… which has (radical) implications on how we should educate people for computer science …

… and on how we apply what we have learnt …

What we want from Parallelism

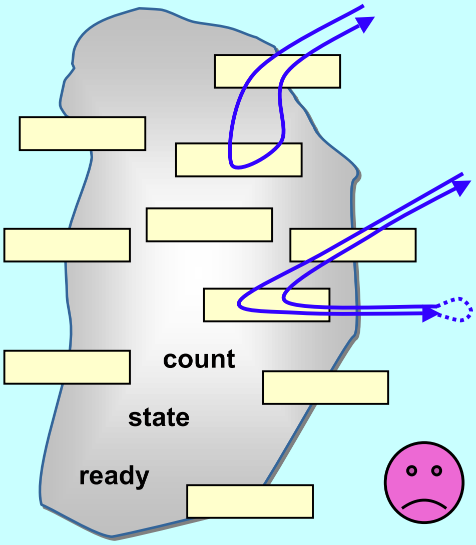
- A powerful tool for **simplifying** the description of systems.
- Performance that spins out from the above, but is **not** the primary focus.
- A model of concurrency that is mathematically clean, yields no engineering surprises and scales well with system complexity.

Objects Considered Harmful

Most objects are dead - they have no life of their own.

All methods have to be invoked by an external thread of control - they have to be *caller oriented* …

… a somewhat curious property of so-called *object oriented* design.

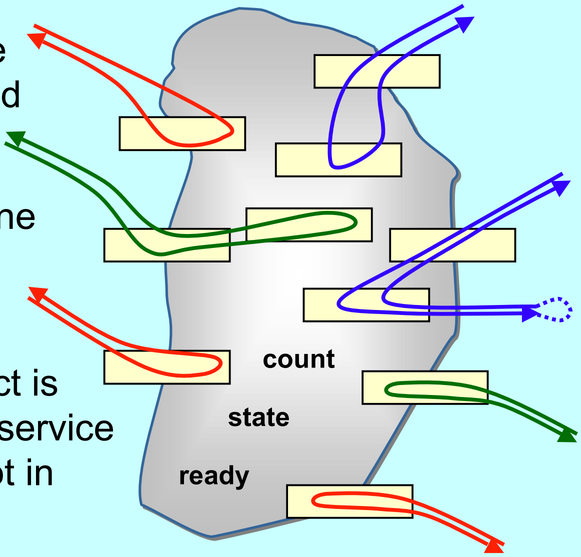


Objects Considered Harmful

The object is at the mercy of *any* thread that sees it.

Nothing can be done to prevent method invocation ...

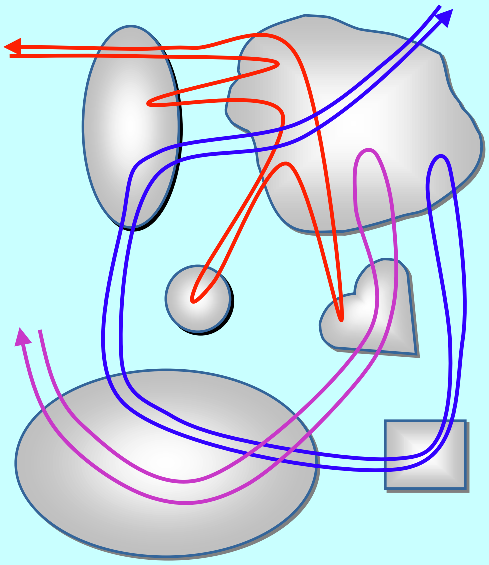
… even if the object is not in a fit state to service it. The object is not in control of its life.



Objects Considered Harmful

Each single thread of control snakes around objects in the system, bringing them to life *transiently* as their methods are executed.

Threads cut across object boundaries leaving spaghetti-like trails, *paying no regard to the underlying structure*.



Java Monitors - CONCERNS

- Easy to learn - but very difficult to apply … safely …
- Monitor methods are *tightly interdependent* - their semantics compose in *complex ways* … the whole skill lies in setting up and staying in control of these complex interactions …
- Threads have no structure … there are no *threads within threads* …
- Big problems when it comes to scaling up complexity …

Java Monitors - CONCERNS

- Almost all multi-threaded codes making direct use of the Java monitor primitives that we have seen *(including our own)* contained race or deadlock hazards.
- Sun’s Swing classes are not thread-safe … why not?
- One of our codes contained a race hazard that did not trip for two years. This had been in daily use, its sources published on the web and its algorithms presented without demur to several Java literate audiences.

Java Monitors - CONCERNS

`<java.sun.com/products/jfc/tsc/articles/threads/threads1.html>`

- ***“ If you can get away with it, avoid using threads. Threads can be difficult to use, and they make programs harder to debug. ”***
- ***“ Component developers do not have to have an in-depth understanding of threads programming: toolkits in which all components must fully support multithreaded access, can be difficult to extend, particularly for developers who are not expert at threads programming. ”***

Java Monitors - CONCERNS

`<java.sun.com/products/jfc/tsc/articles/threads/threads1.html>`

- ***" It is our basic belief that extreme caution is warranted when designing and building multi-threaded applications … use of threads can be very deceptive … in almost all cases they make debugging, testing, and maintenance vastly more difficult and sometimes impossible. Neither the training, experience, or actual practices of most programmers, nor the tools we have to help us, are designed to cope with the non-determinism … this is particularly true in Java … we urge you to think twice about using threads in cases where they are not absolutely necessary …"***

Java Monitors - CONCERNS

- So, Java monitors are not something with which we want to think - certainly not on a daily basis.
- If we have to think with them at all, we want some serious help!
- The first step in getting that help is to build a formal model of what we think is happening …

Communicating Sequential Processes (CSP)

A mathematical theory for specifying and verifying complex patterns of behaviour arising from interactions between concurrent objects.

CSP has a formal, and *compositional*, semantics that is in line with our informal intuition about the way things work.

Claim

Why CSP?

- Encapsulates fundamental principles of communication.
- Semantically defined in terms of structured mathematical model.
- Sufficiently expressive to enable reasoning about *deadlock* and *livelock*. Process semantics are in terms of its *traces*, *failures* and *divergences*.
- Abstraction and refinement central to underlying theory.
- Robust ***and commercially supported*** software engineering tools exist for formal verification.

Why CSP?

- CSP libraries available for Java (**JCSP, CTJ**).

- Ultra-lightweight kernels* have been developed yielding ***sub-hundred-nano-second*** overheads for context switching, process startup/shutdown, synchronized channel communication and high-level shared-memory locks.

- Easy to learn and easy to apply (*WYSIWYG*) …

* not yet available for JVMs

So, what is CSP?

CSP deals with ***processes***, ***networks*** of processes and various forms of ***synchronisation*** / ***communication*** between processes ***(events).***

Processes are active entities - they perform (internal) computations and engage in events. Each process, `P`, is registered for a particular set of events in which it might engage - this set is called its alphabet, `alpha(P)`.

Events are barriers. If *one* process tries to engage in an event, *all* processes (whose alphabets include that event) must engage in it before the engagement will complete.

So, what is CSP?

Channels are special kinds of events, usually in the alphabets of just two processes - the sender and the receiver. Channels communicate values. Each <*channel.value*> pair represents a different event.

Processes may be combined in ***sequence*** (`P;Q`), in ***parallel*** (`P||Q`), by ***internal choice*** (`P⊓Q`) and by ***external choice*** (`P□Q`).

If `a` is an event and `P` is a process, then (`a` → `P`) is a process meaning: ***engage in*** `a`***, then become*** `P`.

`SKIP` is a process that starts, does nothing and terminates.

Java Monitors

The key Java primitives for thread synchronisation are:

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

We will only be concerned with how running threads interact in this paper. We will not be concerned, therefore, with starting threads (`Thread.start`) or waiting for them to terminate (`Thread.join`).

We have no intention of considering the deprecated thread control methods (`Thread.start`, `Thread.suspend` and `Thread.resume`).

Java Monitors

The key Java primitives for thread synchronisation are:

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

We will also not be dealing with `Thread.interrupt` in this paper - nor with the `InterruptedException` that might be thrown by the `wait()` method.

And we're not going to deal with timeouts either!

Nor priorities!!

A CSP Model of Java Monitors

Let ***Objects*** be an enumeration of all Java objects. For any particular system, this can be restricted to just those on which a synchronized action is performed. For example:

***Objects* = {0, 1, 2}**

Let ***Threads*** be an enumeration of all Java threads. For any particular system, this can be restricted to just those threads that are created and started. For example:

***Threads* = {0, 1}**

A CSP Model of Java Monitors

Define the following set of channels:

channel **claim, release, waita, waitb, notify, notifyAll**

: Objects.Threads

This introduces six families of channel - each family is indexed by an object and a thread. For example:

claim.o.t

where **o** is in ***Objects*** and **t** is in ***Threads.***

Modeling the Monitor Interface

Claiming and releasing a monitor lock:

```
synchronized (o) {
  ...  some process P
}
```

This maps to:

```
claim.o.me ⟶ P; release.o.me ⟶ SKIP
```

where `me` is the thread executing the above code.

Note that this also gives us the model for `synchronized` methods.

Modeling the Monitor Interface

Formally, entry and exit to a `synchronized` block or method is modeled by the processes:

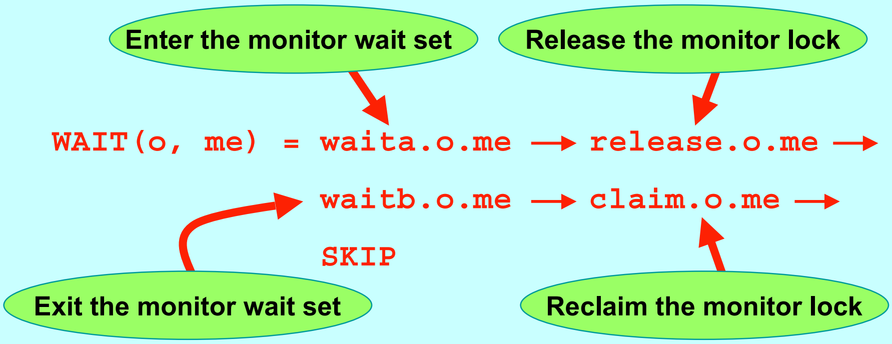
```
STARTSYNC(o, me) = claim.o.me → SKIP

ENDSYNC(o, me)  = release.o.me → SKIP
```

where `o` is the monitor object being synchronised and `me` is the thread doing the synchronising.

Running in parallel with all user processes will be system processes `MONITOR(o)`, one for each object `o` being used as a monitor. This has the above `claim.o.me` event in its alphabet and, if the lock is held by another thread, will refuse to engage in it. That will block entry to the monitor by the `me` thread.

Modeling the Monitor Interface

The `o.wait()` method is modeled by the process:

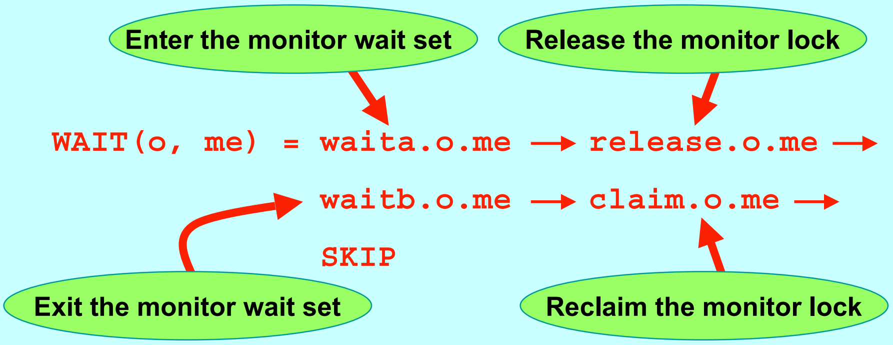

Enter the monitor wait set
Release the monitor lock
Exit the monitor wait set
Reclaim the monitor lock

```
WAIT(o, me) = waita.o.me → release.o.me →
              waitb.o.me → claim.o.me →
              SKIP
```

where `o` is the monitor object whose lock has to be held and `me` is the thread that is doing the waiting.

Modeling the Monitor Interface

The `o.wait()` method is modeled by the process:



Enter the monitor wait set
Release the monitor lock
Exit the monitor wait set
Reclaim the monitor lock

```
WAIT(o, me) = waita.o.me → release.o.me →
              waitb.o.me → claim.o.me →
              SKIP
```

Again, the system process, `MONITOR(o)`, ensures the above semantics.

Modeling the Monitor Interface

The `o.notify()` method is modeled by the process:

```
NOTIFY(o, me) = notify.o.me → SKIP
```

The `o.notifyAll()` method is modeled by the process:

```
NOTIFYALL(o, me) = notifyAll.o.me → SKIP
```

Again, the system process, `MONITOR(o)`, interacts with the above user process actions to arrange the exit from the wait set of some waiting user process (or processes) and its (their) reclaim of the monitor lock.

Summary of the Monitor Interface

```
STARTSYNC(o, me)  = claim.o.me  → SKIP

ENDSYNC(o, me)  = release.o.me  → SKIP

WAIT(o, me)  = waita.o.me  → release.o.me  →
                 waitb.o.me  → claim.o.me  →
                 SKIP

NOTIFY(o, me)  = notify.o.me  → SKIP

NOTIFYALL(o, me)  = notifyAll.o.me  → SKIP
```

… where the blue events above may cause the user threads engaging in them to block (because `MONITOR(o)` has those events in its alphabet and may refuse to engage) .

The 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) || MWAIT(o,{})
```

The `MLOCK(o)` process controls the *locking* of the monitor associated with object `o` - i.e. it deals with `synchronized`.

The `MWAIT(o,{})` process controls the (initially empty) *wait-set* of threads currently stalled on this monitor - i.e. it deals with `wait()`, `notify()` and `notifyAll()`.

The 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) || MWAIT(o,{})
```

The alphabet of `MONITOR(o)` is just the union of the alphabets of its component processes.

This will be the case for all parallel process compositions in this paper.

Locking the Monitor

Each `MLOCK(o)` is basically a binary semaphore. Once claimed by a thread, only a release from that same thread will let it go - meanwhile it refuses all other claims.

If this were all it had to do, it could be simply modelled:

```
MLOCK(o) = claim.o?t → release.o.t →
             MLOCK(o)
```

Where the alphabet of `MLOCK(o)` is:

```
{claim.o.t, release.o.t | t in Threads}
```

Locking the Monitor

However, a constraint in Java is that `o.wait()`, `o.notify()` and `o.notifyAll()` are only allowed if the invoking thread has the monitor lock on `o`.

Therefore, we extend the alphabet of `MLOCK(o)` to:

```
{claim.o.t, release.o.t, waita.o.t,
 notify.o.t, notifyAll.o.t | t in Threads}
```

In its initial *unlocked* state, `MLOCK(o)` *refuses* all the extra events. In its *locked* state, `MLOCKED(o)` *accepts* those extra events suffixed by the locking thread, but these have no impact on its state.

Locking the Monitor

The `MLOCK(o)` process then becomes:

```
MLOCK(o)  =  claim.o?t  ⟶  MLOCKED(o,t)
```

where:

```
MLOCKED(o,t)  =  release.o.t  ⟶  MLOCK (o)
                 □ waita.o.t  ⟶  MLOCKED(o,t)
                 □ notify.o.t  ⟶  MLOCKED(o,t)
                 □ notifyAll.o.t  ⟶  MLOCKED(o,t)
```

So, `MLOCK(o)` prevents *any* `o.wait()` / `o.notify()` / `o.notifyAll()`. `MLOCKED(o)` allows them, but only by the locking thread.

Managing the Wait-Set

The `MWAIT(o,ws)` process controls the *wait-set*, `ws`, belonging to the monitor object `o`. Its alphabet contains just the ***wait*** and ***notify*** events:

```
{waita.o.t, waitb.o.t,

 notify.o.t, notifyAll.o.t | t in Threads}
```

In its initial state, `MWAIT(o,ws)` listens for a `waita.o?t`, `notify.o?t` or `notifyAll.o?t`, from any thread `t`.

It releases a waiting thread, by communicating on some `waitb.o.s` - but only in response to a `notify.o.t` or `notifyAll.o.t`.

Managing the Wait-Set

```
MWAIT(o, ws) =

  waita.o?t → MWAIT(o, ws union {t})

□

  notify.o?t →

    if |ws| > 0 then

       ⊓      waitb.o!s → MWAIT(o, ws minus {s})
    s in ws
    else

      MWAIT(o, {})

□

  notifyAll.o?t → RELEASE (o, ws)
```

Releases *one* thread, chosen non-deterministically

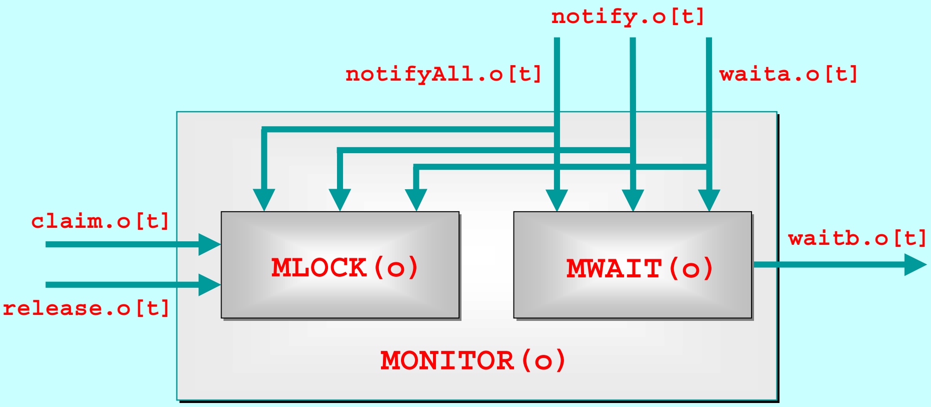
Managing the Wait-Set

```
RELEASE(o, ws) =
    if |ws| > 0 then
        $\sqcap_{s \text{ in } ws}$ waitb.o!s → RELEASE(o, ws minus {s})
    else
        MWAIT(o, {})
```

Releases *all* threads in some non-deterministic order

`RELEASE(o,ws)` follows a `notifyAll.o?t`. It must release all the waiting threads (if any) in some non-deterministic order.

The Monitor Processes

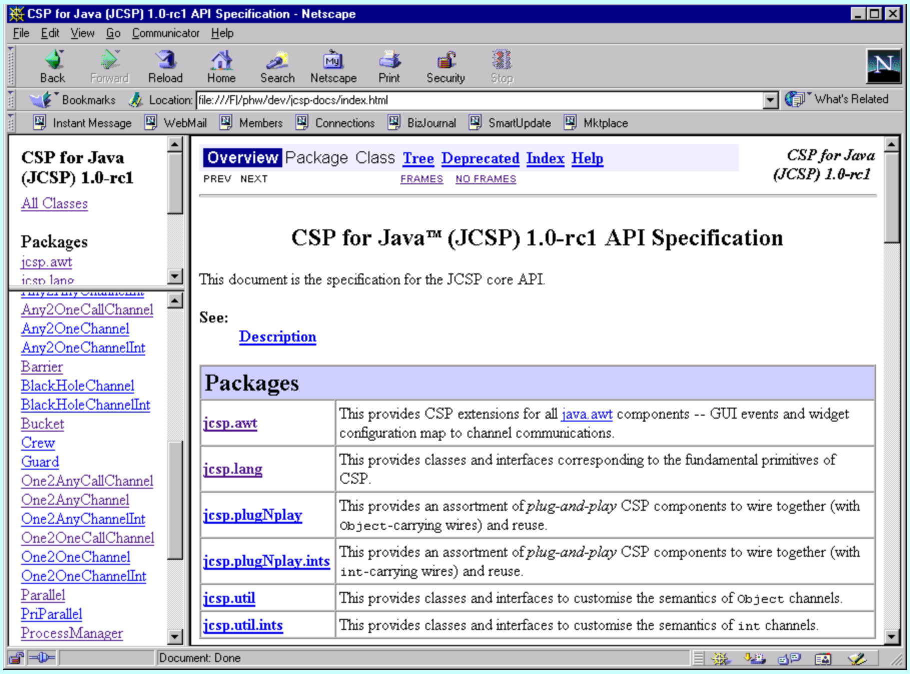

notify.o[t]
notifyAll.o[t]
waita.o[t]
claim.o[t]
release.o[t]
MLOCK(o)
MWAIT(o)
waitb.o[t]
MONITOR(o)

Each arrow represents an *array* of channels, one for each thread, `t`, that synchronizes on this monitor, `o`. The shared channels are broadcasters - both sub-processes must input.

Putting CSP into practice …

JCSP

http://www.cs.ukc.ac.uk/projects/ofa/jcsp/


CSP for Java (JCSP) 1.0-rc1 API Specification - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: file:///F|/phw/dev/jcsp-docs/index.html What's Related
Instant Message WebMail Members Connections BizJournal SmartUpdate Mktplace
Overview Package Class Tree Deprecated Index Help
PREV NEXT FRAMES NO FRAMES
Document: Done

CSP for Java (JCSP) 1.0-rc1

All Classes

Packages

jcsp.awt
jcsp.lang

Any2OneCallChannel
Any2OneChannel
Any2OneChannelInt
Barrier
BlackHoleChannel
BlackHoleChannelInt
Bucket
Crew
Guard
One2AnyCallChannel
One2AnyChannel
One2AnyChannelInt
One2OneCallChannel
One2OneChannel
One2OneChannelInt
Parallel
PriParallel
ProcessManager

CSP for Java (JCSP) 1.0-rc1

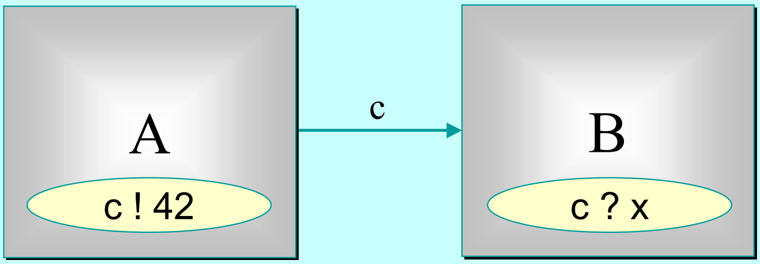
CSP for Java™ (JCSP) 1.0-rc1 API Specification

This document is the specification for the JCSP core API.

See:
Description

Packages	
jcsp.awt	This provides CSP extensions for all java.awt components -- GUI events and widget configuration map to channel communications.
jcsp.lang	This provides classes and interfaces corresponding to the fundamental primitives of CSP.
jcsp.plugNplay	This provides an assortment of *plug-and-play* CSP components to wire together (with `Object`-carrying wires) and reuse.
jcsp.plugNplay.ints	This provides an assortment of *plug-and-play* CSP components to wire together (with `int`-carrying wires) and reuse.
jcsp.util	This provides classes and interfaces to customise the semantics of `Object` channels.
jcsp.util.ints	This provides classes and interfaces to customise the semantics of `int` channels.

Synchronised Communication

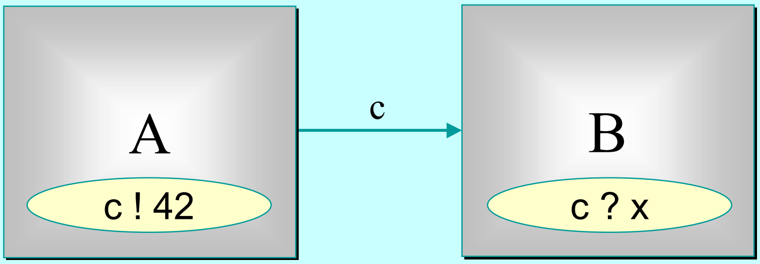


A may write on ***c*** at any time, but has to wait for a *read*.

B may read from ***c*** at any time, but has to wait for a *write*.

A (c) || B (c)

Synchronised Communication



Only when both ***A*** and ***B*** are ready can the communication proceed over the channel ***c***.

A (c) || B (c)

The JCSP One2OneChannel

```
public class One2OneChannel {

  private Object channelHold;     // data in transit

  private boolean channelEmpty;   // sync flag

  ...  public sync Object read ()

  ...  public sync void write (Object mess)

}
```

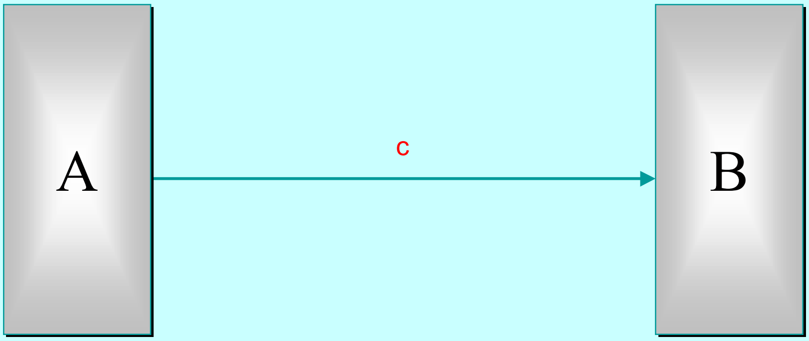
The JCSP One2OneChannel

```
public synchronized Object read ()
 throws InterruptedException {

  if (channelEmpty) {        // first to rendezvous

    channelEmpty = false;
    wait ();                 // wait for the writer
    notify ();               // schedule writer to finish

  } else {                   // second to rendezvous

    channelEmpty = true;
    notify ();               // schedule waiting writer

  }
  return channelHold;
}
```

The JCSP One2OneChannel

```
public synchronized void write (Object mess)
 throws InterruptedException {

  channelHold = mess;
  if (channelEmpty) {         // first to rendezvous

    channelEmpty = false;
    wait ();                  // wait for the reader

  } else {                    // second to rendezvous

    channelEmpty = true;
    notify ();                // schedule waiting reader
    wait ();                  // let reader regain lock

  }
}
```

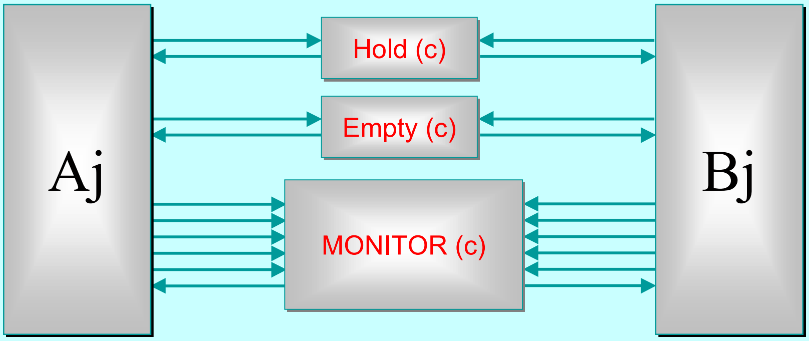
Synchronised Communication



THEOREM:

For any processes A and B, the above CSP network …

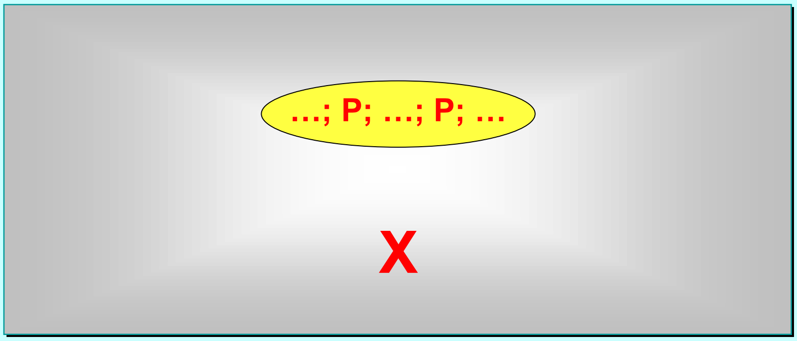
Synchronised Communication



THEOREM (cont.):

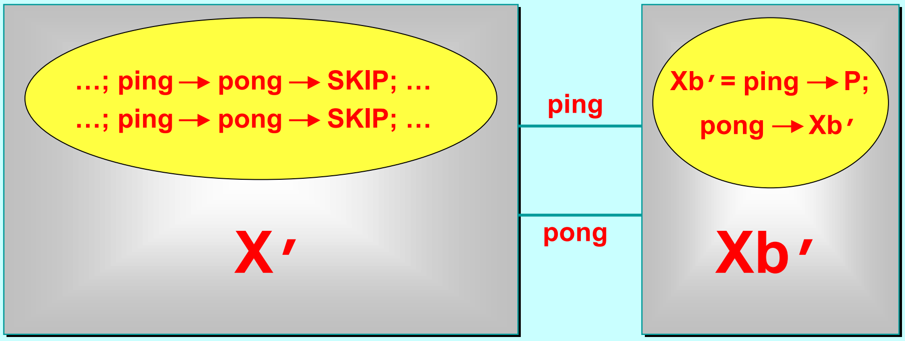
… is equivalent to this one!

Parallel Introduction



Gratuitous introduction of parallel processes can be done anywhere. Suppose `X` contains a serial process including one or more instances of `P` …

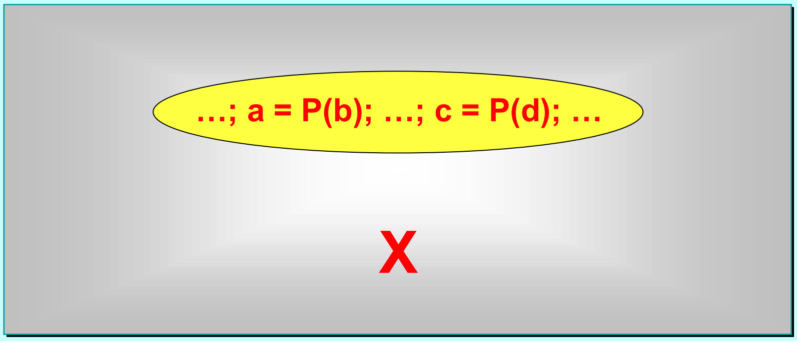
Parallel Introduction


…; ping → pong → SKIP; …
…; ping → pong → SKIP; …
X′
ping
pong
Xb′ = ping → P;
pong → Xb′
Xb′

Then, **X** is equivalent to the parallel composition of the above two processes. **X′** is **X** with all instances of **P** replaced by **(ping → pong → SKIP)** . **Xb′** is a *buddy* process that just performs **P** on its behalf.

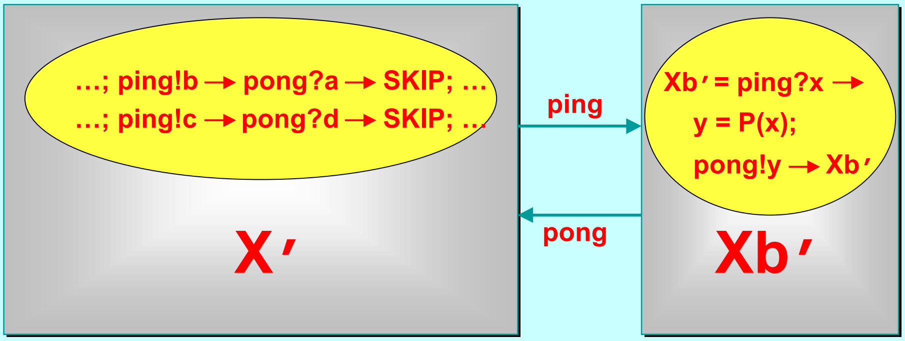
Simple CSP algebra

Parallel Introduction



If the process being delegated accesses and modifies the state of **X** …

Parallel Introduction



…; ping!b → pong?a → SKIP; …
…; ping!c → pong?d → SKIP; …

X′

ping

pong

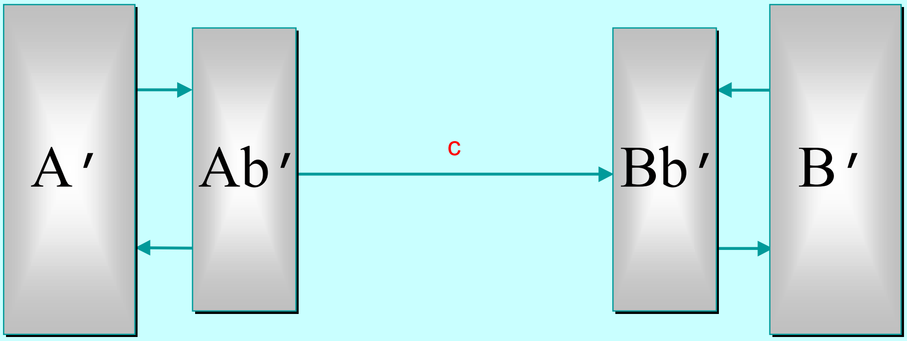
Xb′ = ping?x →
y = P(x);
pong!y → Xb′

Xb′

Then, **ping** and **pong** become channels through which the state and update results are passed, as well as retaining their synchronisation roles between `X′` and `Xb′`.

Simple CSP algebra

Synchronised Communication



A′

Ab′

c

Bb′

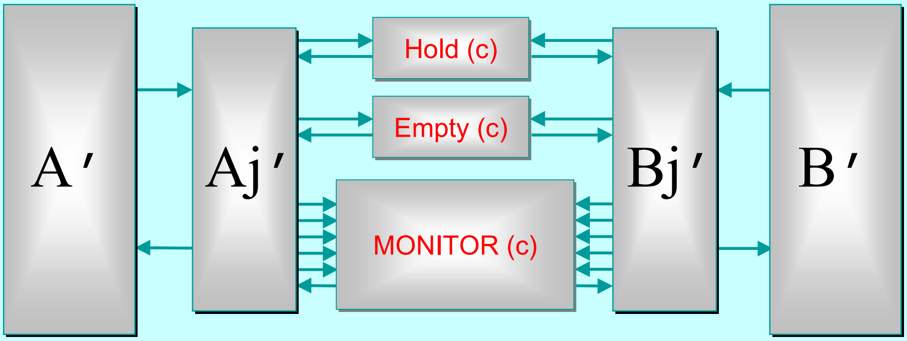
B′

LEMMA:

`(A′ || Ab′) = A` *and* `(B′ || Bb′) = B`

Parallel Introduction

Synchronised Communication

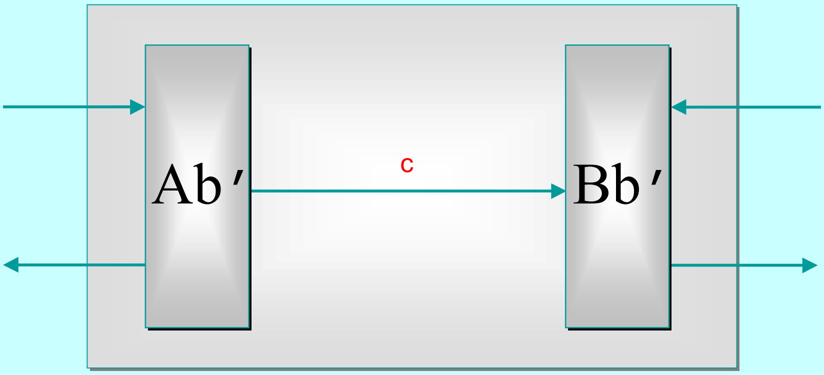

A′
Aj′
Hold (c)
Empty (c)
MONITOR (c)
Bj′
B′

LEMMA:

`(A′ || Aj′) = Aj` *and* `(B′ || Bj′) = Bj`

Parallel Introduction

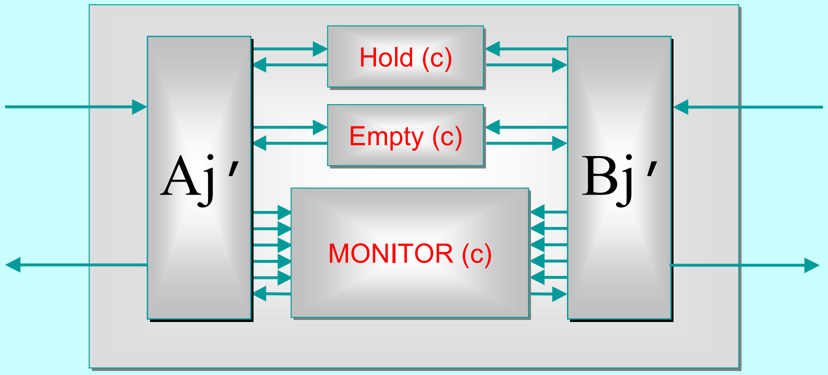
Synchronised Communication



LEMMA:

The above process (that hides the internal channel `c` and just provides two *ping-pong* interfaces) is equivalent to …

Synchronised Communication



LEMMA: **Proof by FDR** ☺

… the above process (that also hides all internal channel **c** and just provides two *ping-pong* interfaces) .

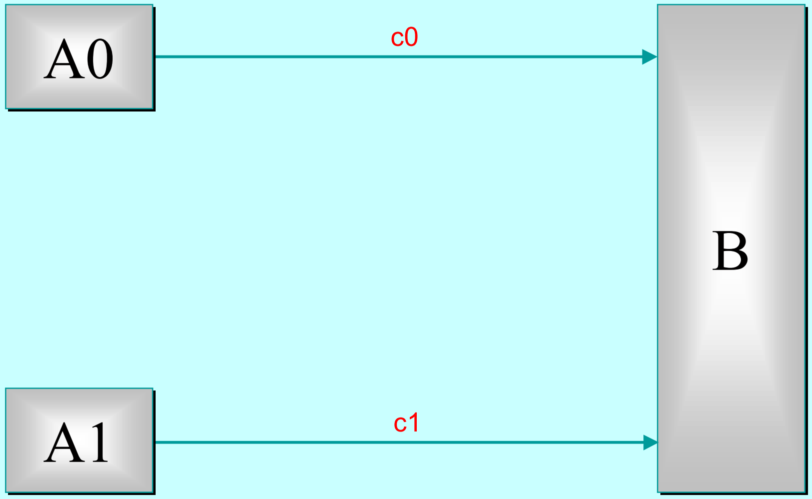
Summary - I

- A CSP model for the Java monitor mechanisms (`synchronized`, `wait`, `notify`, `notifyAll`) has been built.
- This enables *any* Java threaded system to be analysed in CSP terms - e.g. for formal verification of freedom from deadlock/livelock.
- Confidence gained through the formal proof of correctness of the JCSP channel implementation:
  - a JCSP channel is a non-trivial monitor - the CSP model for monitors transforms this into an even more complex system of CSP processes and channels;
  - using FDR, that system has been proven to be a refinement of a single CSP channel and *vice versa* - **Q.E.D.**

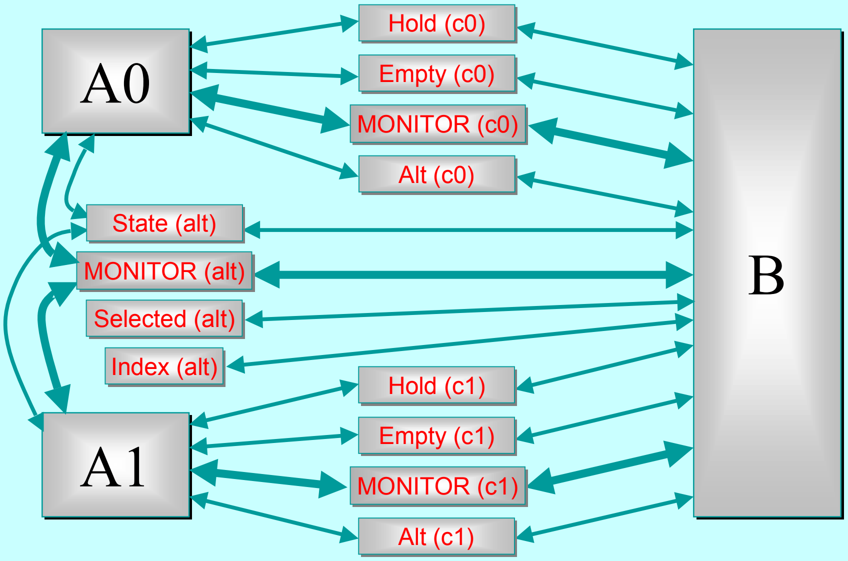
Summary - II

- Extending the model to deal with *re-acquisition* of a monitor lock by a thread already holding it *(easy)*.
- Extending the model to deal with *interrupts* and the exception raised by **`wait()`** - *(OK, but not so easy)*.
- Analysing published Java multithreaded codes and looking for trouble!
- Verifying (hopefully) the implementation of the rest of the JCSP primitives - especially those concerning *Alternation*, where earlier undetected race hazards were a real shake to our confidence …

ALTing Communication



ALTing Communication



A0

A1

B

Hold (c0)

Empty (c0)

MONITOR (c0)

Alt (c0)

State (alt)

MONITOR (alt)

Selected (alt)

Index (alt)

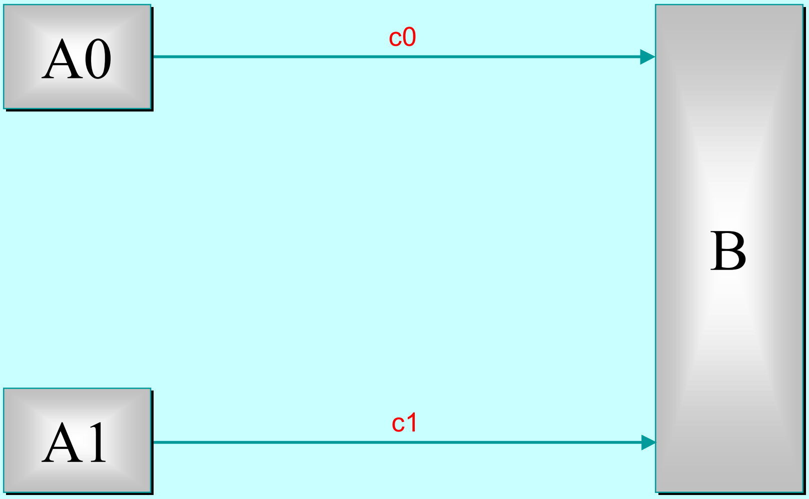
Hold (c1)

Empty (c1)

MONITOR (c1)

Alt (c1)

ALTing Communication



A0

A1

B

c0

c1

Summary - III

- Using *parallel introduction* in the same way gives the model checker something on which to bite.
- FDR, after working through around 48000 states, confirms that the two networks are equivalent.
- FDR revealed the deadlock trace in the CSP network representing the original implementation within seconds. That deadlock had taken 2 years to reveal itself in practice!
- This has only verified a *2-way* **ALT**. FDR could probably manage a *3-way* one … but an *n-way*?
- Nevertheless, a huge confidence boost for JCSP.

Final Thought …

- A CSP model of Java's multithreading puts it on a solid engineering foundation.
- Systems can be analysed for refinement, equivalence, deadlock, livelock and race hazards.

"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?"