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

Using Java for Parallel Computing: JCSP versus CTJ, a Comparison

Nan C. Schaller
Computer Science Dept., Rochester Institute of Technology
102 Lomb Memorial Drive, Rochester, NY 14623-5608, USA
ncs@cs.rit.edu

Gerald H. Hilderink
University of Twente, P.O.Box 217, 7500 AE Enschede
Control Laboratory, The Netherlands
g.h.hilderink@el.utwente.nl

Peter H. Welch
Computing Laboratory, University of Kent
Canterbury, UK CT2 7NF
P.H.Welch@ukc.ac.uk

Abstract. Java provides support for concurrent and parallel programming through threads, monitors and its socket and Remote Method Invocation (RMI) classes. However, there have been many concerns expressed about the way in which this support is provided, e.g., [1][2], citing problems such as improper implementation of monitors and difficulty of programming with threads. Hoare's *Communicating Sequential Processes* (CSP) [3][4][5] model fully specifies thread synchronization and is based on processes, compositions, and channel communication. It provides a mathematical notation for describing patterns of communication using algebraic expressions and contains formal proofs for analyzing, verifying and eliminating undesirable conditions, such as race hazards, deadlocks, livelock, and starvation. Two independent research efforts provide a CSP based process-oriented design pattern for concurrency implemented in Java: *Communicating Sequential Processes for Java* (JCSP) [6] and *Communication Threads in Java* (CTJ) [7]. In this paper, we compare these two packages, looking at the philosophy behind their development, their similarities, their differences, their performance, and their use.

1. Introduction

Java has been touted as the "write once, run anywhere" programming language. One of the long-standing issues in concurrency, parallel programming and high performance computing has been lack of portability. Thus, many researchers are exploring the use of Java in this field. But, there are many problems with using Java for parallel computing: Java's RMI does not provide a flexible enough scheme to be used for many concurrent and parallel programming paradigms. Java's synchronisation primitives are too low level, unsafe and difficult to use correctly. For example, the monitor-threads model provided by Java, while easy to understand in its primitives, proves difficult to apply with confidence [1][2] in any system above a modest level of complexity. Numerous warnings in Java

textbooks (and on some of Sun's web pages) emphasise the difficulties of multi-threading (race hazards, deadlock, livelock and process starvation) and recommend getting involved only as a last resort. In addition, Java's performance is not impressive when compared to other languages. There has been some performance improvement with Just-In-Time (JIT) compilers, but it is still not close to what one can achieve using languages such as C and C++. However, concurrency is too powerful and, indeed, too simple an idea to be set aside. With a better handle, it can simplify both the design and the implementation of most complex systems, as well as boost performance. And, researchers would like to program concurrency in Java – particularly for internet, interactive, embedded and high performance computing – for all the reasons that Java is so popular.

A recent partial web search for packages that use Java for developing tools that handle concurrency resulted in this partial alphabetical list, categorized by type:

- Alternative languages: Titanium [8]
- Active Agents – Concordia [9], IBM Aglets [10], JAFMAS [11], JATLite [12], Kafka used in Pathwalker [13], Mole [14], Odyssey [15], and ProActive PDC [16]
- Cluster Computing: Albatross [17] and JavaNOW [18][19]
- Corba-like: HORB [20], JacORB [21], Jorba [22], sJava [23] and Voyager [24]
- Dataflow: Dataflow Java [25]
- Distributed Shared Memory: JUMP [26] and MultiJav [27]
- Java Virtual Machine (JVM) Changes: cJVM [28]
- Linda-like or Shared Memory – Jada [29], Javaspaces [30], JOMP[31], and TSpaces [32]
- Message Passing: CSP-OZ to Java[33], CTJ [7], Infospheres [34], and JCSP [6]
- Metacomputing: Byanihan [35], DOGMA [36], and Javelin [37]
- MPI: HPJava [38], JavaMPI [39], jmpi [40], and mpiJava [41]
- PVM: JPVM [42] and jPVM [43]
- RMI Changes : JavaParty[44], Manta [45], and NinjaRMI [46]
- (New) Synchronization Primitives: Jsync [47] and PtolemyII [48]
- Utilising OS and Hardware resources: Jaguar [49]

To weed through all of these packages is difficult, especially as there are often problems with packages that are available on the web. Many of them represent research projects for completing academic degrees. These may not have a life after that degree is completed. Several of the packages are available for Java 1.1 only. It is not clear if this is because research or maintenance has stopped, or because later versions of Java are sufficiently different that it is difficult to get the package to work correctly under them. Some packages are implemented fully in Java, others provide a Java binding to a previously existing package. Some of the packages are only discussed in reports and are not publicly available. Others are implementations that are sold commercially, which means that one may have to pay to even try them out. And, there is always the problem that the web is a moving target, i.e., the links that one finds today may have moved or removed tomorrow.

The search also yielded organizations whose goals are to find ways to use Java effectively for high performance computing (e.g., Java Grande [50]) and to create real-time and embedded systems specifications for Java (e.g., the Real-Time Java Group [51] and the J-consortium [52]). Some efforts are dedicated to make Java packages that one can use to write programs that are formally verifiable (e.g., CTJ [7] and JCSP [6]).

This paper will focus on this last category, which not only provides formal verification, but more importantly, enables complex parallel applications to be built, ones that can be scaled up without losing control. Our web search yielded only these two publicly available

packages. These two independent research efforts have introduced the CSP model into Java through sets of classes implemented on top of its monitor support: Java Communicating Sequential Processes (JCSP) from the University of Kent, UK [6] and Communication Threads in Java (CTJ) [7] from the University of Twente, NL. In this paper, we compare these packages, looking at the philosophy behind their development, their similarities, their differences, their performance, and their use.

2. A(n Almost) Common Philosophy

The first Java Threads Workshop [53] organised by Java users (under the umbrella of the World **occam** and Transputer User Group) was held at the University of Kent, England, in September, 1996. This workshop focused on design and performance issues for threaded applications in Java. The creators of the current versions of both CTJ and JCSP were intimately involved with the workshop and indeed, have communicated frequently during the development of their packages.

Both libraries are based on the philosophy that concurrent behavior from the objects in a system ought to be the normal expectation – not something difficult that is added in as an advanced feature to improve user response times or other performance indicators. Concurrency should provide:

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

Java's built-in monitor concepts score badly on the above [1][2]. Both CTJ and JCSP use instead a model based on *Communicating Sequential Processes* (CSP) [3][4][5]. CSP is a mathematical theory for specifying and verifying complex patterns of behavior arising from interactions between concurrent objects. CSP has a formal and compositional semantics that lines up with ones informal intuition about the way things work.

So, CSP deals with *processes*, *networks* of processes and various forms of *synchronization* and *communication* between them. A network of processes is also a process – so CSP naturally accommodates layered structures *(networks of networks*). Both CTJ and JCSP incorporate these ideas.

CTJ and JCSP have a common core in the base algorithms for CSP primitives, but have different emphasis and design philosophy. CTJ is one of the ongoing projects culminating from Gerald Hilderink's Ph.D. research under the tutelage of André Bakkers and Jan Broenink at the University of Twente. The impetus behind the development of CTJ was to create a Java package for creating real-time and embedded software using the CSP model. In particular, to develop control software for 20-sim [54], a modeling and simulation program, and that would compete with dSPACE [55] (a commercial product that purportedly *"revolutionizes your development processes with real-time systems for rapid control prototyping, production code generation, and hardware-in-the-loop tests"*). Thus, the important motivations behind CTJ (as well as JCSP) are:

- to make things reasonably safe but not too restrictive;
- to make compromises so as not to introduce unreasonable inefficiencies;
- to specify thread synchronisation and scheduling behaviour that is platform independent and that enables real-time capabilities (CTJ specific).

The motivation behind JCSP was to generate a real practical product - an alternative and sane API for concurrency in Java.. JCSP aims to combine a clean approach to CSP capabilities with an acceptance of the peculiarities of Java. Hence, it shares with CTJ the first two of the above bullets. The syntactic and semantic differences in its API were the result of slightly different trade-offs being chosen (between safety and efficiency) and independent teams. In fact, ideas that were initially introduced in JCSP were later taken up by CTJ – *and vice-versa*. These independent ideas may not have taken place if the development efforts had been explicitly merged..

JCSP was born as a result of an incredibly good final year undergraduate project by Paul Austin at the University of Kent, who came up with the idea of the `Parallel` and the `AWT` processes. His professor, Peter Welch, felt that the project was too good to leave, picked it up and continues its development.

3. Similarities

To understand the similarities between CTJ and JCSP, one must know more about CSP as both libraries have a common core in the base algorithms for implementing CSP primitives. We use a simple producer/consumer example to illustrate the similarities in the packages. The Producer sends an integer message to the Consumer, who will simply receive the message and print out its contents. This activity is diagrammed in Figure 1. A complete implementation of the Producer-Consumer Process Network is provided in sections 3.4 in CTJ and in 3.5 for JCSP.

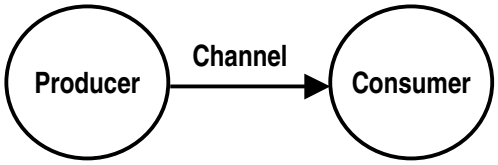

Figure 1. Producer-Consumer Process Network

3.1 Processes

A CSP *process* is a component that encapsulates data structures and algorithms for manipulating that data. Both its data and algorithms are private. The outside world can neither see that data nor execute those algorithms.

Each process is alive, executing its own algorithms on its own data. Processes interact solely via CSP synchronizing primitives, such as channels, not by calling each other's methods. Objects implementing those primitives form the CSP interface to a process.

Figure 1 shows two processes, `Producer` and `Consumer`, connected by a CSP channel labeled `Channel` that provides the conduit over which the two processes may interact. In CTJ and JCSP, each process is an instance of a class implementing the `csp.lang.Process` or `CSProcess` interface, respectively, e.g.:

```
public interface csp.lang.Process {
  public void run ();
}
```

```
public interface CSProcess {
  public void run ();
}
```

In both, the behavior of a process is defined by the implementation of its `run()` method. Figures 3 and 4 display an implementation for the `CTJ_Producer` and `CTJ_Consumer` classes, while Figures 6 and 7 for the `JCSP_Producer` and `JCSP_Consumer` classes.

Each shows the implementation of the appropriate interfaces. For example, in CTJ:

```
class CTJProducer implements Process {   // import csp.lang.Process
  ...
}
```

and in JCSP:

```
class JCSP_Producer implements CSProcess {
  ...
}
```

There are some "rules" that pertain to the construction of a CSP-type process in CTJ and JCSP:

- Any public constructors or mutator methods for these processes must install the shared synchronisation objects into the private fields, i.e., channels. They may also, of course, initialise other private state information.
- Any public accessor/mutator methods (simple sets and gets) may be invoked only when this process is not running. They should be the responsibility of a single process only, usually the process that constructed this one.
- That constructing process is also responsible for triggering the public `run()` method that kicks this one into life, usually in parallel with some other constructed processes (see Section 4.3). The private support methods are invoked only by each other and by the `run()` method and express the live behaviour of this process.

There are also some properties that pertain to each of these processes:

- A process instance may have several lives but these must, of course, be consecutive. Different instances of the same process class may be alive concurrently
- When a process is running, it is in sole charge of its private fields. Its thread of control never leaves the process and no foreign threads can enter. No other processes can inspect or interfere with those fields.
- Changes of state may be requested by other processes, e.g. through channel communication, but this process is at liberty to refuse even to listen to such requests. Both sides must actively cooperate to exchange information, so neither can be surprised when this happens.

This last property of localized semantics, preserved under parallel composition, is a major reason why CSP-concurrent design is so manageable.

3.2 Synchronizing Channels

The simplest form of process interaction is synchronised message-passing along channels. The simplest form of channel is zero-buffered and point-to-point. Such channels correspond directly to our intuition about a wire connecting two hardware components.

In Figure 1, `Producer` and `Consumer` are processes and `Channel` is a channel connecting them. Channels, like wires in an electronic circuit, have no capacity to hold information, being only media for transmission. To avoid undetected loss of data, channel communication is synchronized. This means that if `Consumer` transmits before `Producer` is

ready to receive, then `Consumer` will block. Similarly, if `Producer` tries to receive before `Consumer` transmits, `Producer` will block. When both are ready, data is transferred directly from the state space of `Consumer` into the state space of `Producer`.

Both CTJ and JCSP have special channel classes to handle integer messages. Figures 3 and 6 show a channel *write* of an integer message in the `CTJ_Producer` and `JCSP_Producer` classes, respectively, while Figures 4 and 7 show a channel *read* of that integer message in the `CTJ_Consumer` and `JCSP_Consumer` classes, respectively. The appropriate channel declarations using CTJ are:

```
ChannelOutput_of_Integer outChannel;      // In CTJ_Producer
ChannelInput_of_Integer inChannel;        // In CTJ_Consumer
```

and using JCSP:

```
ChannelOutputInt outChannel;              // In JCSP_Producer
ChannelInputInt inChannel;                // In JCSP_Consumer
```

The communication of the integer message is accomplished using CTJ by:

```
n.value = 100;                            // n is a csp.lang.Integer
outChannel.write (n);                     // In CTJ_Producer
inChannel.read (n);                       // In CTJ_Consumer
```

and using JCSP by:

```
outChannel.write (100);                   // In JCSP_Producer
value = inChannel.read ();                // In JCSP_Consumer
```

Note that the set of CSP synchronization primitives that defines the interface between a process and its environment is not part of any Java interface. Instead, it must be plugged into each process via public constructors or mutator methods when the process is not running, i.e., before or in between runs when it is safe to extract information from a process.

3.3 Networks

A process-oriented design consists of layered networks of processes. A network is simply a parallel composition of processes connected through a set of passive synchronisation objects (e.g., wires) and is itself a process.

Each process fulfills a contract with its environment that specifies not only what functions it performs, but how it is prepared to synchronize with that environment to obtain information and deliver results.

Note that a process does not interact directly with other processes, rather only with the channels to which it is connected. This is a familiar form of component interface, certainly to hardware engineers, and one that allows considerable flexibility and reuse.

Figure 1 represents a (possibly layered) network of processes. The diagram represents an overall process that consists of a parallel composition of two processes, the `Producer` and the `Consumer`, that are connected by a `Channel`. The implementation of this overall process is shown in Figure 2 for CTJ (CTJ_PCMain.java) and in Figure 5 (JCSP_PCMain.java) for JCSP. Notice that both of these processes have the common components of a channel object of the appropriate type, a process that is a parallel composite of their Producer and Consumer processes and a statement `par.run()` – that causes the parallel composition to begin executing.

3.4 The Producer-Consumer Network implemented using CTJ

This section contains the complete implementation of the Producer-Consumer Network in CTJ. Assuming the CTJ library has been downloaded and the CLASSPATH has been correctly set, to execute the CTJ network one need only compile the three pieces of code in Figures 2-4 (`javac *.java`), and then execute CTJ_PCMain (`java CTJ_PCMain`).

```
import csp.lang.*;
import csp.lang.Process;      // override java.lang.Process
import csp.lang.Integer;      // override java.lang.Integer

public class CTJ_PCMain {

  public static void main (String[] args) {
    new CT_JPCMain ();
  }

  public CTJ_PCMain () {

    // Create channel object
    final Channel_of_Integer channel = new Channel_of_Integer ();

    // Create parallel construct with a list of processes
    Process par =
      new Parallel (
        new Process[] {
          new CTJ_Producer (channel),
          new CTJ_Consumer (channel)
        }
      );

    // Run parallel composition
    par.run ();
  }

}
```

Figure 2: CTJ_PCMain.java

```
import csp.lang.*;
import csp.lang.Process;      // override java.lang.Process
import csp.lang.Integer;      // override java.lang.Integer

class CTJ_Producer implements Process {

  ChannelOutput_of_Integer outChannel;

  public CTJ_Producer (ChannelOutput_of_Integer out){
    outChannel = out;
  }

  public void run () {
    Integer n = new Integer ();
    n.value = 100;
    outChannel.write (n);
  }

}
```

Figure 3: CTJ_Producer.java

```
import csp.lang.*;
import csp.lang.Process;      // override java.lang.Process
import csp.lang.Integer;      // override java.lang.Integer

class CTJ_Consumer implements Process {

  ChannelInput_of_Integer inChannel;

  public CTJ_Consumer (ChannelInput_of_Integer in) {
    inChannel = in;
  }

  public void run () {
    Integer n = new Integer ();
    inChannel.read (n);
    ...  do something with n
  }

}
```

Figure 4: CTJ_Consumer.java

3.5 The Producer-Consumer Network implemented using JCSP

This section contains a complete implementation using JCSP. Again, assuming the JCSP library has been downloaded and the `CLASSPATH` has been correctly set, to execute the JCSP network one need only compile the three pieces of code in Figures 5-7 (`javac *.java`), and then execute JCSP_PCMain (`java JCSP_PCMain`).

```
import jcsp.lang.*;

public class JCSP_PCMain {

  public static void main(String[] args) {
    new JCSP_PCMain ();
  }

  public JCSP_PCMain () {

    // Create channel object
    final ChannelInt channel = new One2OneChannelInt ();

    // Create parallel construct with a list of processes
    CSProcess par =
      new Parallel (
        new CSProcess[] {
          new JCSP_Producer (channel),
          new JCSP_Consumer (channel)
        }
      );

    // Run parallel composition
    par.run ();
  }

}
```

Figure 5: JCSP_PCMain.java

```
import jcsp.lang.*;

class JCSP_Producer implements CSProcess {

  ChannelOutputInt outChannel;

  public JCSP_Producer (ChannelOutputInt out) {
    outChannel = out;
  }

  public void run () {
    outChannel.write (100);
  }
}
```

Figure 6: JCSP_Producer.java

```
import jcsp.lang.*;

class JCSP_Consumer implements CSProcess {

  ChannelInputInt inChannel;

  public JCSP_Consumer (ChannelInputInt in) {
    inChannel = in;
  }

  public void run () {
    int n = inChannel.read ();
    ...  do something with n
  }

}
```

Figure 7: JCSP_Consumer.java

3.6 Alternation – Choosing Between Events

A crucial CSP operator is *choice* or *selection, i.e.,* the ability of a process to wait for one of several events to occur, reacting to whichever shows up first and choosing between them if many are pending.

Both packages provide a passive mechanism for doing the waiting, i.e., there is no active polling for events, and three ways for resolving any offered choice (arbitrary, user-prioritized and fair). This is discussed in more detail in Section 4.2.

3.7 More Similarities

Both packages add CSP primitives to Java. Both have Java classes that implement CSP constructs for `Sequential`, `Parallel`, `PriParallel`, `Alternative` (with *arbitrary*, *prioritised* or *fair* choice), `PriAlternative`, `Skip`, `Stop`, `Guards`, `Process`, and the **occam** `Tagged Protocol` (called `Channel_of_Reference` in CTJ). Both have facilities to create all types of channels: *one-to-one*, *any-to-one*, *one-to-any*, *any-to-any*, buffered and unbuffered, and *call channels*.

Note that call channels provide a standard Java interface for channel communication. Any Java interface can be turned into a call channel. This gives an object-oriented familiar means of inter-process communication that contains all the benefits of CSP channel communications, i.e., both sides have to be involved so that both know what is happening. Those things cannot happen to process's state without that process's involvement is a key benefit from CSP. Hence, separate consideration of each process is possible. That is not the case with the threads/monitor model.

4. Class Differences

CTJ and JCSP differ in the APIs provided for the basic primitives, reflecting different design ideas, and in the extensions each of them pursues (CTJ to provide flexible support for real-time constraints, JCSP to provide support for general concurrent programming, such as GUI/graphics and modelling). We discuss some of these differences below.

4.1 Channels and Message Types

The reader most likely has already noted that the interfaces used to send the integer message in our example looked a bit different in the two libraries. There are actually several differences in this area that we indicate in the Table 1. Some are subtler than one might expect and are noted below.

Notes on Table 1:

1. In JCSP the correct use of *one-to-one*, *one-to-any*, *any-to-one* and *any-to-any* channels is not automatically checked and is the user's responsibility. The programmer can misuse these channels, e.g., by connecting two readers or writers to a *one-to-one* channel. In this case, the semantics are undefined and anything can happen, including correct channel behavior, an incorrect termination of a channel read or write, or incorrect message delivery. This means that that if the user constructs a *one-to-o*ne channel, it is the user's responsibility to dedicate it to connecting just two processes. CTJ channels are all *any-to-any*, so this issue does not arise.

2. The CTJ channel passes messages by value normally, but passing messages by reference is possible. The JCSP channel passes messages, other than int, by reference. CTJ's approach of passing messages by value encourages the reuse of objects. Once a message object has been sent, the sender process may reuse that message object. The CTJ creators have determined that passing messages by value does not unnecessarily fragment memory, and makes less intensive use of the garbage collecting because of this reuse. This method of message passing works as well for distributed non-shared memory systems.

That CTJ passes messages by value means that communication over a soft (shared memory) channel and a hard (external) channel connecting separate memory JVMs have the same semantics. This consistency is very nice. JCSP external channels, when completed, will almost certainly have a message-by-value semantics, which would different from its internal channels. This is not so nice. But JCSP's internal channels are fast and no less secure than CTJ's message-by-value ones; both are insecure and rely on design rules to make them safe.

CTJ's message passing does provide some support in this area, but it is not complete. One problem is that it is not enough to forget about the object that has just sent down a channel, which CTJ does provide. One must also forget about any other objects for which a reference is held and for which the sent object also hold references! Getting automatic

control of this would be an important breakthrough. What is required is the ability to "message-by-reference" for internal channels, but where the sender loses the reference, and references to anything referred to by the thing just sent. That would be fast and safe. Sadly, Java is not up to this! What is needed is a language, perhaps a derivative of Java, that takes aliasing issues seriously, probably from the point of view of concurrency, although getting this right brings big benefits to purely serial programming. (Peter has a Ph.D. student, Tom Locke, working on this.)

Table 1. Channel Related Differences

	CTJ	**JCSP**
Types of Channels (See Note 1)	Any-to-Any Others could be added.	One-to-One, One-to-Any Any-to-One, Any-to-Any BlackHole
Types of Messages	The types of all messages are essentially Objects, but specific channel classes are provided for message types Any, Boolean, Byte, Character, Double, Float, Integer, Long, Object, Poison, Reference, and Short. In addition, CTJ provides wrapper classes for Java primitive data types that allow their values to be overwritten	The type of message may be Object or the Java primitive data type `int`. Message types for other Java primitive data types could easily be added (but there seems little demand).
How messages are passed (See Note 2)	Messages are normally passed by value, but may be passed by reference.	Messages of type `int` are passed by value. Messages of type `Object` are passed by reference. The user may choose to make a copy of the message before sending.
Hardware to Hardware channels	Channels may be used for external communication, but these must be constructed using the `Linkdriver` class. `Linkdrivers` provide the protocol of the hardware dependent data transfer. Some sample link drivers are provided.	Hardware to hardware communication is available automatically on SMP hardware only. For any other situation, the user must extend the appropriate channel classes.
GUI/Graphics	Supports a link to standard Java AWT/SWING components using a special linkdriver.	Contains a set of special JCSP processes that provide channel interfaces to AWT components.
Real-Time Support (See Note 3)	Provided through built-in real-time kernel connected with the channels that schedules their own threads in a well-specified and real-time fashion.	Only real-time on top a of a real-time operating system.

3. The CTJ CSP channel concept deals with single- and multi-processor environments and also takes care of the real-time priority scheduling requirements. Careful examination of priority and scheduling has determined that priority scheduling should be attached to the communicating channels rather than to the processes themselves. A priority based `Parallel` construct, `PriParallel`, was developed in association with channels for composing processes, hiding threads and priority indexing from the user. This approach simplifies the use of priorities for the object-oriented paradigm. Moreover, the notion of scheduling is no longer connected to the operating system but has rather become part of the application.

4.2 Specification and Implementation of Alternatives

Appendix A (Section 8) shows a CTJ implementation of the simple process network, diagrammed in Figure 8, that consists of two Producers and one Consumer. Because the Producers may operate at different speeds, it is an ideal occasion for the Consumer to use an `Alternative` to determine which has data available for reading at any one time. Appendix B (Section 9) shows the implementation of the same network using JCSP. It is hoped that the reader will refer to these implementations while considering the differences in the implementation of `Alternative` in the two packages. Notice that only the consumer processes changes significantly from our previous example.

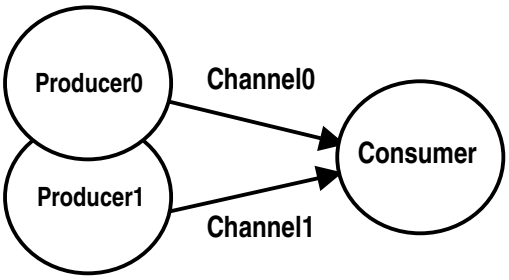

Figure 8. Two Producer-Consumer Process Network

In CTJ there are two classes for selection, the `Alternative` and `PriAlternative`. As `PriAlternative` is a subclass of `Alternative` and `Alternative`'s `select()` method is polymorphic, both classes then are specifying a single `select()` method. An `Alternative` then consists of a `select()` method, a process or a guard that enables nesting with other `Alternatives` as in **occam** and CSP.

```
final Alternative alt =              // this version of
  new Alternative (                  // the Alternative
    new Guard[] {                    // can only be used
      new Guard (inChannel[0]),      // with its select()
      new Guard (inChannel[1])       // method
    }
  );

Integer n = new Integer ();          // n is an Object

for (int i = 0; i < 20; i++) {       // for example

  final int index = alt.select ();   // wait for a channel
  inChannel[index].read (n);         // read from selected channel

  ...  do something with n

}
```

Figure 9: CTJ Alternative set up for use with a `select()` – see Appendix A

There are two ways one can use the `Alternative` in CTJ: with a `select()` method as shown in Figure 9 or as a compositional construct as shown in Figure 10. For explicit selection, the `select` *method* can be used; for a more structural and compositional alternation, the guarded *process* construct can be used. The `run()` method of that guarded process is executed by the `Alternative` *process* with almost the same performance as

switching on a returned `select` index – there is the overhead only of its method invocation. In addition, one can easily add, remove and insert guards to the `Alternative` construct. And, with proper nesting of `Alternative` and `PriAlternative` constructs, the user may create the equivalent of fair and priority selection using either the select or compositional constructs.

```
final Integer n = new Integer ();    // n is an Object

final Process alt =
  new Alternative (
    new Guard[] {
      new Guard (inChannel[0], new Process() {
        public void run () {
          inChannel[0].read (n);
          ...  do something with n
        }
      }),
      new Guard (inChannel[1], new Process() {
        public void run () {
          inChannel[1].read (n);
          ...  do something with n
        }
      })
    }

  );

for (int i = 0; i < 20; i++) {      // for example

  alt.run ();      // make the selection and run the response

}
```

Figure 10: CTJ Alternative set up with *guarded processes* – see Appendix A

In JCSP, the input channels are already `Guard`s (they `extend` them) – so no new `Guard` array is needed. In the `Alternative`, as shown in Figure 11, JCSP uses the same `select()` technique as CTJ, but takes a different approach to prioritising that choice. Instead of subclassing, `Alternative` offers the following set of methods:

- `select()`, returns an *arbitrary* choice;
- `priSelect()`, returns a *prioritised* choice (in order of the `Guard` array);
- `fairSelect()`, returns a *fair* choice when repeatedly invoked.

```
final Alternative alt = new Alternative (inChannel);

for (int i = 0; i < 20; i++) {

  final int index = alt.fairSelect ();  // or priSelect() or select()
  final int n = inChannel[index].read ();

  ...  do something with n

}
```

Figure 11: JCSP Alternative set up – see Appendix A

Note that the channels are of type `AltingChannelInputInt` in this example. In JCSP, the `select()` approach only was taken for ALTing because one of its objectives was to make compromises with Java so as not to introduce unreasonable inefficiencies. In **occam** (and CSP), `ALT` (and the choice operator) is a constructor that takes *guarded processes* as arguments. CTJ follows this principle, although it also allows for the `select()` approach. But *processes* in Java, however they are implemented, are not as trivial to define as they are in **occam** and CSP. They are *instances of classes* that need to be able to access and alter global items, rather than simple fragments of code.

So, if the `Alternative` constructor were to take processes as arguments, the programmer has to go through the whole Java process of creating those object instances. This is a bit obscure and expensive, both in the syntactic overhead required and, unless handled carefully, in run-time. So, the JCSP `Alternative` just focuses on an array of `Guards` – which can include any user-defined mix of channel inputs, CALL channel accepts, timeouts and SKIPs – and returns an index to the one selected. That index can then be used in the normal sequential flow of control offered by Java, often through a `switch` statement on the index. This introduces no syntactic and run-time overhead of inner classes in the response to that `Alternative` guard.

There is something else for which a user needs to beware. Because CTJ has only *any-to-any* channels, safety errors will arise should someone try to use an `Alternative` on a channel that happens to be used in a design with *many* receivers. This is documented as a constraint that the CTJ user must honour. Disregarding this constraint leads to code whose semantics are undefined. Tools could be developed that check for such illegal *xxx-to-xxx* relationships. In JCSP, *one-to-any* and *any-to-any* channels may not be used as ALT guards for the same reason. However, breaking this rule in JCSP is trapped at compile-time.

4.3 PriParallel Implementations

Both CTJ and JCSP support prioritising processes that are running in parallel. JCSP's priority parallel is limited to whatever priorities the underlying JVM implementation provides and the semantics of those underlying JVM priorities. CTJ has a much more aggressive approach to this, as is needed for real-time applications. `PriParallel` constructs may be nested within other `PriParallel` constructs, allowing an unlimited level of priorities that the underlying kernel then handles within its priority scheduling duties. In the future, JCSP may support its API with the CCSP version of the KroC [57] kernel that will support multiple priorities and external drivers properly.

4.4 Additional Features

Both packages provide additional classes: In CTJ, these include:

- `Thread` – specifies a process that assigns a new thread to a process and continues immediately. Each instance of `Thread` can be used in all composition constructs.
- `TimeSlicer` – can be used to time slice between processes of the same priority. A process that must be appended to a dispatcher implements it. It will only time slice between processes with lower priority than the `TimeSlicer` process itself.
- `Timer` – provides timeout guards (similar to **occam** `TIMER`s).
- *Call channels* – channels with a *method* interface.
- `Version` - print the version number of the CTJ library.

In JCSP, these include:

.

- `Barrier` – enables barrier synchronisation between a set of processes.
- `Bucket` – a non-deterministic version of barrier synchronisation [58][59].
- `Crew` – provides a *Concurrent Read Exclusive Write* (CREW) lock for synchronising fair and secure access to a shared resource.
- `Timer` – provides timeout guards (similar to **occam** `TIMER`s).
- *Call channels* – channels with a *method* interface.
- `ProcessManager` – enables a `CSProcess` to be spawned concurrently with the process doing the spawning. This is similar to CTJ's `Thread`.
- *Plug-and-Play* components similar to that used in the **occam** course material. These components emphasise code reuse.
- Utilities to customise the semantics of object `channel`s and `int`s, e.g., buffered channels of several types.

4.5 Package Accessibility and Appearance

Both packages are works in progress and continue to evolve. The CTJ creators have put most of their efforts into the programming end and less in the product presentation end. Example code is available, but it is not (yet) nicely packaged and an interested party must work to find them.

The JCSP documentation is vast and full of mini-tutorials, e.g., those on the CALL channels. Considerable effort has been spent on documenting the library – as much as, or possibly more than, on its coding.

5. Performance

The performance of both packages is not any worse than programming in Java directly, but this is still disappointing. For instance, communication time with JCSP under Java Development Kit (JDK) 1.1.x clocks in at over 100 microseconds per context switch on a 500 MHz PIII. Under JDK1.2.2, that reduces to around 23 microseconds. So, things are improving. However, the KRoC **occam** (from the same research team) context switch on the same machine is around 140 *nanoseconds*. One approach to obtaining better performance might be to create a JVM binding to the KRoC kernel. This is under consideration – indeed, work is under way [61]. In the future, JCSP may support its API with the CCSP version of the KRoC kernel that supports multiple priorities and external drivers properly, and with context switch times also measured in nanoseconds.

As parallel processes in both CTJ and JCSP are mapped onto Java threads, that in turn are mapped by modern JVMs to native threads, all the processors can be used on an SMP machine and real speedup can be obtained with the right kind of problem. Absolute performance times, though, are hard to predict on a multi-user system, such as a Sun 450 server, since they depend on machine loading at the time the benchmarks are run. On a single user SMP, such as a multi-processor PC, results are more predictable.

One importance of CTJ is its real-time kernel that schedules its own threads in a well-specified and predictable fashion. CTJ does not require an operating system and can be used on bare control hardware. Because CTJ has control over its own threads, the spread in its performance is quite small. JCSP, on the other hand, is (currently) only real-time on top of a real-time operating system.

Both packages function well under JDK1.2. However, there are some JCSP demonstration applets downloadable alongside the package itself. Some of these allow the user to interact with and control animated graphics, where the graphics are computed and rendered in real-time. Rendering images in JDK1.2 is horribly slow in comparison to JDK1.1.8, or any earlier JDKs down to 1.1.5. So, these demonstrations work much better under JDK1.1.x than JDK1.2.x. All animated graphics applications/applets suffer from this in JDK1.2; i.e., it is not a JCSP problem. The recently released JDK1.3 has not been tried yet, but the beta versions had not solved this problem. In addition, whether the applets work at all seems to be entirely hardware dependent. They all work fine from browsers running under Windows™, but some functions disappear mysteriously under Solaris™ or Linux – so much for *"write once, run anywhere"*!

The creators of both packages are considering a JVM binding approach to improve the performance. In CTJ, Gerald is considering creating a binding to his CSP versions of C and C++, which are copies of the ideas behind CTJ and which are very fast compared to CTJ. As stated before, Peter is likewise considering a JVM binding that accesses the KRoC kernel. This also outperforms JCSP.

6. Reports of Use

Both packages are in use internationally and have been well received. CTJ has been used for course work at universities in such places as South Africa (Rhodes), the United States (Utah State), and in England (Westminster) for classes involving concurrency, parallel computing and real-time computing. In some cases, exercises included using CTJ's linkdrivers to achieve true parallelism, i.e., to execute on multiple machines. Students from other universities have reportedly used CTJ successfully for their individual projects. Inquiries have been received from industries such as Philips and Oracle as well.

JCSP has experienced similar results. It is used for course work at universities in places such as England (Kent), the United States (Colgate and Rochester Institute of Technology), and in Italy (Pisa). It has been used for courses that teach concurrency and parallel computing. While JCSP does not support the use of multiple computers directly other than on SMP platforms, additional constructors for channel classes have, in at least one instance, been used to facilitate programs to execute in a truly parallel fashion. JCSP is also featured in section 4.5 of the second edition of Doug Lea's book: *Concurrent Programming in Java: Design Principles and Patterns* [56].

7. Conclusions

Both CTJ and JCSP are viable Java class libraries providing a base range of CSP primitives. Both libraries enable multithreaded systems to be designed, implemented and reasoned about entirely in terms of CSP synchronising primitives, e.g., channels and events, and constructors, e.g., parallel and choice. This allows 20 years of theory, design patterns with formally proven good properties, such as the absence of race hazards, deadlock, livelock and thread starvation, tools supporting those design patterns, education and experience to be deployed in support of Java multithreaded applications and eventually, high performance computing.

The CSP channel concept implemented in these libraries is a natural way to use multithreading without being troubled with Java threads programming. The thread administration is completely handled by the CSP channel. In fact, the CSP addition provides a concurrent object-orientated and process-oriented programming tool without the

additional burden of programming threads. The resulting code is not only easy to program and understand, it is also safe to use because the rules of CSP, which both packages are based on, guarantee the correct interaction between concurrent processes. The programmer need not be concerned with the formal mathematical theory, because CSP is mature, well-founded, and some powerful model checking tools **Error! Reference source not found.**, based on that mathematics, are available for when they are needed.

CTJ contains a real-time kernel that schedules its own threads in a well-specified and real-time fashion. CTJ does not need an operating system and can be used on bare control hardware.

JCSP, which provides an additional rich set of extensions and a package providing CSP process wrappers giving a channel interface to all Java AWT widgets and graphics operations, is extensively (javadoc)umented and includes much teaching material.

Both are in use today internationally and an extended lifetime is anticipated. However, it may require some help from the Java creators for these packages to have good performance for high performance computing – to make them competitive with C and C++ packages. However, both are considering a JVM binding approach to improve the performance; CTJ with its sister libraries for C and C++ and JCSP with the KRoC kernel.

References

[1] P. H. Welch. Java Threads in Light of occam/CSP. In *Architectures, Languages and Patterns, WoTUG-21*, pp. 259-284, IOS Press (Amsterdam), April 1998.

[2] P. Brinch-Hansen. Java's Insecure Parallelism. *ACM SIGPLAN Notices*, 34, 4, pp. 38-45, April 99.

[3] C. A. R. Hoare. Communicating Sequential Processes, *Communications of the ACM*, 21-8, pp. 666-677, August 1978.

[4] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, London, UK, 1985.

[5] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[6] P. H. Welch and P. D. Austin. The JCSP Home Page. http://www.cs.ukc.ac.uk/projects/ofa/jcsp/**,** 1999.

[7] G. H. Hilderink. The CTJ (Communicating Threads in Java) Home Page. http://www.rt.el.utwente.nl/javapp, 2000.

[8] The Titanium Home Page. http://http.cs.berkeley.edu/projects/titanium/, 1999.

[9] The Concordia Home Page. http://www.meitca.com/HSL/Projects/Concordia/, 2000.

[10] The IBM Aglets Workbench. http://www.trl.ibm.co.jp/aglets/, 1999.

[11] A. Galan. The JAFMAS Home Page. http://www.ececs.uc.edu/~abaker/JAFMAS, 1998.

[12] The JATLite Home Page. http://java.stanford.edu/java_agent/html/, 1998.

[13] The Pathwalker Home Page. http://www.fujitsu.co.jp/hypertext/flab/free/paw/index.html**,** Jujitsu, 2000.

[14] J. Baumann, et al. The Mole Home Page. http://mole.informatik.uni-stuttgart.de/, 1999.

[15] The Odyssey Web Site: http://www.genmagic.com/agents/.

[16] The ProActive Team. The ProActive PDC Home Page. http://www-sop.inria.fr/sloop/javall/, 1999.

[17] H. Bal, et al. The Albatross Home Page. www.cs.vu.nl/albatross/, 2000.

[18] The JavaNOW Source Page. http://www.jhpc.cs.depaul.edu/general/resources.html.

[19] The JavaNOW Home Page. http://www.plexobject.com/software/javanow/javanow.html, PlexObject Solutions.

[20] The HORB Developers Group. The HORB Home Page. http://ring.etl.go.jp/openlab/horb/, 2000.

[21] G. Brose, et al. The JacORB Home Page. http://www.inf.fu-berlin.de/~brose/jacorb/, 2000.

[22] R. Turner. The Jorba Home Page. http://jorba.castle.net.au/.

[23] C. Petitpiere. The sJava: synchonous Java Home Page. http://ltisun9.epfl.ch/sJava/, 1999.

[24] The Voyager Product Page. http://www.objectspace.com/products/prodVoyager.asp, 2000.

[25] G. Lee and J. Morris. Dataflow Java: Implicitly Parallel Java. In *Proceedings of the Fifth Australasian Computer Architecture Conference*, Canberra, pp. 42-50, February 2000.

[26] K. A. Hawick, et al. *Java Tools and Technologies for Cluster Computing*. University of Adelaide DHPC Technical Report DHPC-077, Available at http://www.dhpc.adelaide.edu.au/reports/077/abs-077.html, November, 1999.

[27] X. Chen and V. H. Allan. The MultiJav Home Page. http://www.cs.usu.edu/~allanv/Pubs/pdpta/pdpta.html, 1999.
[28] cJVM: A Cluster-Aware JVM. http://www.haifa.il.ibm.com/projects/systech/cjvm.html, IBM, 2000.
[29] D. Rossi. The Jada Home Page. http://www.cs.unibo.it/~rossi/jada/, 1996.
[30] The Javaspaces™ Technology Page. http://java.sun.com/products/javaspaces/, Sun Microsystems, 2000.
[31] J. M. Bull and M. E. Kambites. JOMP -- an OpenMP-like Interface for Java. To appear in *Proceedings of the ACM 2000 Java Grande Conference*, June 2000. Available from http://www.epcc.ed.ac.uk/research/publications/conference/acm2k.ps.gz.
[32] Tspaces Project Page. http://www.almaden.ibm.com/cs/TSpaces/, IBM, 2000.
[33] C. Fischer. Combination and implementation of processes and data: from csp-oz to java. Ph.D. thesis. University of Oldenburg, January 2000, available from http://theoretica.Informatik.UniOldenburg.DE/~fischer/cspoz2java.ps.
[34] K. M. Chandy, et al. Catech Infospheres Project. http://www.infospheres.caltech.edu/releases/index.html, 1999.
[35] L. Sarmenta. The Byanihan Home Page. http://www.cag.lcs.mit.edu/bayanihan/, 1999.
[36] DOGMA Source Code. http://zodiac.cs.byu.edu/DOGMA/, 2000.
[37] P. Capella, et al., The Javelin 2.0 Home Page. http://javelin.cs.ucsb.edu/, 2000.
[38] B. Carpenter. The HPJava Home Page. http://www.npac.syr.edu/projects/pcrc/mpiJava/index.html, 2000.
[39] V. S. Getov. The JavaMPI Info Page. http://perun.hscs.wmin.ac.uk/CSPE/software.html, 1998.
[40] K. Dincer. jmpi information accessible under "research" at http://www.ceng.metu.edu.tr/~kdincer/, 2000.
[41] B. Carpenter. The mpiJava Home Page. http://www.npac.syr.edu/projects/pcrc/mpiJava/ mpiJava.html, 2000.
[42] A. Ferrari. The JPVM Home Page. http://www.cs.virginia.edu/~ajf2j/jpvm.html, 1999.
[43] The jPVM (a.k.a. JavaPVM) Home Page. http://www.chmsr.gatech.edu/jPVM/, 1998.
[44] B. Haumacher. JavaParty. http://wwwipd.ira.uka.de/JavaParty/, 1999.
[45] R. vanNieuwpoort. The Manta Home Page. http://www.cs.vu.nl/~rob/manta/, 2000.
[46] M. Welsh. NinjaRMI: A Free Java RMI Introduction and Tutorial. http://www.cs.berkeley.edu/~mdw/proj/ninja/nijarmi.html, 1999.
[47] K. Knizhnik. The Jsync source link. http://www.ispras.ru/~knizhnik/, 2000.
[48] The Ptolemy II Home Page. http://ptolemy.eecs.berkeley.edu/ptolemyII/, 2000.
[49] M. Welsh, The Jaguar (Java Access to Generic Underlying Architectural Resources) Home Page. http://www.cs.berkeley.edu/~mdw/proj/jaguar/, 2000.
[50] The Java Grande Forum Home Page. http://www.javagrande.org/, 2000.
[51] The Real-Time for JavaTM Experts Group Home Page. http://www.rtj.org/, 1999.
[52] The J Consortium™ Home Page. http://www.j-consortium.org/, 2000.
[53] Java Threads Workshop. http://wotug.ukc.ac.uk/parallel/groups/wotug/java/, 1996.
[54] 20-sim Home Page. http://www.rt.el.utwente.nl/clp/products/20sim30.htm, 2000.
[55] *The dSPACE. Home Page*. http://www.dspace.de/, 2000.
[56] D. Lea. *Concurrent Programming in Java (Second Edition): Design Principles and Patterns. The Java Series*, Addison-Wesley, section 4.5, 1999.
[57] P. H. Welch, et al. The KroC Home Page. http://www.cs.ukc.ac.uk/projects/ofa/kroc/, 2000.
[58] P. H. Welch, D. C. Wood, Higher Levels of Process Synchronisation, in A. Bakkers (ed.), Parallel Programming and Java - Proceedings of WoTUG-20, pp. 104-129, IOS Press, Netherlands, April 1997.
[59] J. Kerridge, P. H. Welch, D. C. Wood, Synchronisation Primitives for Highly Discrete Event Simulations, in R. H. Sprague Jnr. (ed.), *Proceedings of the 32nd. Hawaii International Conference on System Sciences (HICSS-32)*, IEEE Computer Society Press. January 1999.
[60] Formal Systems (Europe) Ltd.: *FDR2 Failures-Divergence-Refinement Manual*, http://www.formal.demon.co.uk/FDR2.html, 1997.
[61] J. Moores, Native JCSP – the *CSP for Java* Library with a Low-Overhead CSP Kernel, in Communicating process Architectures 2000 - Proceedings of WoTUG-20, pp. 263-274, IOS Press, Netherlands, September 2000.

Appendix A: The Two Producer-Consumer Network Implemented Using CTJ

This network, diagrammed in Figure 8, may be executed in the manner as detailed in Section 3.4 for the single producer and consumer. This appendix contains the complete listings. The sections pertinent to `Alternative`s are highlighted in `CTJConsumer`.

// file **CTJ_Producer.java**

```
import csp.lang.*;
import csp.lang.Process;      // override java.lang.Process
import csp.lang.Integer;      // override java.lang.Integer

class CTJ_Producer implements Process {

  final private ChannelOutput_of_Integer outChannel;

  public CTJ_Producer (ChannelOutput_of_Integer out){
    outChannel = out;
  }

  public void run () {
    Integer n = new Integer();
    for (int i = 0; i < 10; i++) {
      n.value = 100 + i;
      outChannel.write (n);
    }
  }

}
```

// file **CTJ_Consumer.java (1)**

```
import csp.lang.*;
import csp.lang.Process;      // override java.lang.Process
import csp.lang.Integer;      // override java.lang.Integer

class CTJ_Consumer implements Process {

 final private ChannelInput_of_Integer inChannel[];

  public CTJ_Consumer (final ChannelInput_of_Integer in[]) {
    inChannel = in;
  }

  public void run () {

    final Alternative alt =             // this version of
      new Alternative (                 // the Alternative
        new Guard[] {                   // can only be used
          new Guard (inChannel[0]),     // with its select()
          new Guard (inChannel[1])      // method
        }
      );

    final Integer n = new Integer ();   // n is an Object

    for (int i = 0; i < 20; i++) {

      final int index = alt.select ();  // wait for a channel
      inChannel[index].read (n);        // read from the selected channel

      System.out.println ("from channel " + index + " -->  " + n);

    }

  }

}
```

// file **CTJ_Consumer.java (2)**

```
// Note: this version uses guarded processes when constructing its Alternative.
// It is the same as version (1) above except for the following run() method:

  public void run () {

    final Integer n = new Integer ();    // n is an Object

    final Process alt =
      new Alternative (
        new Guard[] {
          new Guard (inChannel[0], new Process() {
            public void run () {
              inChannel[0].read (n);
              System.out.println ("from channel 0 -->  " + n);
            }
          }),
          new Guard (inChannel[1], new Process() {
            public void run () {
              inChannel[1].read (n);
              System.out.println ("from channel 1 -->  " + n);
            }
          })
        }
      );

    for (int i = 0; i < 20; i++) {
      alt.run ();    // this makes the selection and runs the response
    }

  }

}
```

// file **CTJ_PCMain.java**

```
import csp.lang.*;
import csp.lang.Process;      // override java.lang.Process
import csp.lang.Integer;      // override java.lang.Integer

public class CTJ_PCMain {

  public static void main (String[] args) {

    // Create the channel objects

    final Channel_of_Integer channel [] = new Channel_of_Integer[2];
    channel[0] = new Channel_of_Integer();
    channel[1] = new Channel_of_Integer();

    // Create the network of processes and run it

    new Parallel (
      new Process[] {
        new CTJ_Producer (channel[0]),
        new CTJ_Producer (channel[1]),
        new CTJ_Consumer (channel)
      }
    ).run ();

  }

}
```

Appendix B: The Two Producer-Consumer Network Implemented Using JCSP

This network, diagrammed in Figure 8, may be executed in the manner as detailed in Section 3.5 for the single producer and consumer. This appendix contains the complete listings. The sections pertinent to `Alternative`s are highlighted in `JCSP_Consumer`.

```
////////////////////////////////////////////// file JCSP_Producer.java

import jcsp.lang.*;

class JCSP_Producer implements CSProcess {

  final private ChannelOutputInt outChannel;

  public JCSP_Producer (ChannelOutputInt out){
    outChannel = out;
  }

  public void run () {

    for (int i = 0; i < 10; i++) {
      outChannel.write (100 + i);
    }

  }

}


////////////////////////////////////////////// file JCSP_Consumer.java

import jcsp.lang.*;

class JCSP_Consumer implements CSProcess {

 final private AltingChannelInputInt[] inChannel;

  public JCSP_Consumer (final AltingChannelInputInt[] in) {
    inChannel = in;
  }

  public void run () {

    final Alternative alt = new Alternative (inChannel);

    for (int i = 0; i < 20; i++) {

      final int index = alt.fairSelect ();  // or priSelect() or select()
      final int n = inChannel[index].read ();

      System.out.println ("from channel " + index + " --> " + n);

    }

  }

}
```

```
////////////////////////////////////////////// file JCSP_PCMain.java

import jcsp.lang.*;

public class JCSP_PCMain {

  public static void main (String[] args) {

    // Create the channel objects

    final One2OneChannelInt[] channel = One2OneChannelInt.create (2);

    // Create the network of processes and run it

    new Parallel (
      new CSProcess[] {
        new JCSP_Producer (channel[0]),
        new JCSP_Producer (channel[1]),
        new JCSP_Consumer (channel)
      }
    ).run ();

  }

}
```