Communicating Threads for Java[TM]

Gerald Hilderink, Jan Broenink and André Bakkers

University of Twente, P.O.Box 217, 7500 AE Enschede

Control Laboratory, The Netherlands

g.h.hilderink@el.utwente.nl

Nan C. Schaller

Rochester Institute of Technology

102 Lomb Memorial Drive, Rochester, NY 14623-5608, USA

ncs@cs.rit.edu

DRAFT, REV. 5, October 2000

for CTJ version 0.9 revision 17

(corrections and comments appreciated)

Abstract. While the Java[1] thread model provides support for multithreading within its language and runtime system, the Java synchronization and scheduling strategy is poorly specified and has unsatisfactory real-time performance. This is because Java delegates thread synchronization and scheduling to the underlying operating system, which results in different behavior on different operating systems. This does not support Sun Microsystem's claim that Java is system independent, i.e., that the programmer may "write once, run everywhere". In this paper, we present *Communicating Threads for Java (CTJ)* (**http://www.rt.el.utwente.nl/javapp**), a comprehensive specification of a new thread model for Java that provides system independent scheduling behavior with real-time capabilities and high order synchronization constructs.

1 Introduction

Java's basic thread model is derived from traditional multithreading concepts. Literature about threads and thread synchronization [1,2] deals mostly with operating system, or low-level, concepts of multithreading. In addition, design patterns for concurrent programming using threads [3] address implementation rather than conceptual issues. The freedom and flexibility provided by most thread application programming interfaces (API's) [1,4,5] increase the danger of creating undesirable conditions, such as race hazards, deadlock, livelock, and starvation. Thus, a programmer must reason differently about thread synchronization for each synchronization construct used in an application, by carefully applying a variety of rules, guidelines, and design patterns. Thus, threads are low-level components that cause a major increase in the complexity of writing and testing code. Analyzing a multithreaded program and debugging thread states when using a thread API can also be extremely difficult. The theory of

---

[1] Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. *Communicating Threads for Java* is independent of Sun Microsystems, Inc.

Communicating Sequential Processes (CSP) [8, 9, 10] specifies fundamental synchronization constructs, based on processes, compositions, and channel communication. CSP 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. This model has proven successful for creating concurrent software for real-time and embedded systems [13].

This paper presents the *Communicating Threads for Java* (CTJ) package which implements the CSP model, i.e., processes, compositions and channels, in Java. Its thread patterns are simpler and more reliable than the thread patterns in the Java thread model. The CTJ package provides a small set of design patterns that are sufficient for concurrent programming in Java. An important advantage of CTJ is that the designer or programmer has a rich set of rules or guidelines available that help eliminate undesirable conditions during design and implementation phases.

This paper does not discuss the mathematics of CSP, but rather presents the use of the CSP model for programming in Java. An extensive knowledge of the theory of CSP is not required to understand its concepts. The CTJ package provides a bridge between the theory and the application of CSP in Java.

2 Java threads versus processes in CSP (and CTJ)

The terms *thread* and *process* are closely related. A thread is a stream of control that consists of the processor state and the stack space, e.g., the registers, the instruction pointer, and the stack pointer. A process encapsulates its data and methods, in the much the same way that objects do, but a process also encapsulates one or more threads of control. In other words, assigning a separate thread of control to a passive object creates an active object and turns it into a process.

In CSP and hence in CTJ, a process embraces an active object whose instructions are executed by a single thread of control that is encapsulated within the process. (For the rest of this paper, we will use the term *process* to refer to a CSP/CTJ process.) A process is *never* able to interfere with the work of any other process. Processes may start other processes, but they *may not* directly control them. Distinct process address spaces prohibit processes from altering the states of other processes. This is because only "read-only" variables may be shared between processes; variables that may be modified directly by a process are always local to that process and are never shared between processes. Only that single thread can update the contents of its variables, and therefore, there is no danger of race-hazards.

Cooperation between processes is established using channel communication. For example, the contents of a variable local to one process, and needed by another, must be communicated to the other over a channel. Channels themselves are primitive, passive objects that processes can send or receive messages over.

Furthermore, the behavior of Java threads is poorly specified and strongly depends on the behavior of the underlying operating system. Therefore, threads may behave differently on different Java virtual machines (JVMs), whereas processes will behave the same way on all JVMs. Debugging processes then becomes far less difficult than tracing thread behavior as when using the Java thread model directly. Now, debugging an application means only debugging its processes.

2.1 Synchronization primitives

In Java, more than one thread may be assigned to a single object. Threads that can operate on shared data simultaneously must be synchronized to prevent race-hazards that can result in

corrupt data or in invalid states. The user can control each thread using a variety of methods, each of which must be used in a specific way. This set of methods, found in the *java.lang.Thread* and *java.lang.Object* classes [5], provides a basic, flexible multi-threading model. For example, to create a monitor [5], the `synchronized` clause can be used to create a critical region around shared data that allows only one thread to enter that region at a time. The `wait()/notify()` methods are needed to perform conditional queuing of threads within that critical region. This means that methods that access the monitor cannot be designed individually, but rather must be designed in concert with other methods that access that monitor. The monitor as a synchronization construct is expensive for a single threaded application. Furthermore, it is not always trivial to determine if methods or regions need to be synchronized. There are design patterns [3] that can be used to help solve this problem, but they make programs more complex than is necessary.

In CSP and CTJ, channels provide the synchronization. Channels are intermediate passive objects that sit between processes and take care of synchronization, scheduling, and message delivery. Furthermore, the programmer is freed from having to use complicated synchronization and scheduling constructs.

CSP channels are unbuffered and are synchronized according to the *rendezvous principle* [8,9]; the sending process waits until the receiving process is ready, or the receiving process waits until the sending process is ready before communication takes place. CTJ also provides buffered channels, which are far more efficient than using buffering processes. An important message is that using channels is simpler than using the monitor construct and thinking in terms of processes is more abstract and easier for engineers than thinking in terms of threads, as will be illustrated in the next section.

2.2   State transitions

Only a single thread is executed at any time on a processor. A multiprocessor system with *n* processors can therefore execute *n* threads simultaneously. However, a single processor system may, in fact, execute multiple threads using scheduling and timesharing [1,2] techniques that slice the main thread into multiple sub-threads.

As a thread or process executes, it changes state. The states and the state transitions of threads and those of processes may not necessarily be represented by the same state transition diagrams, even if they are executed on the same system. We will briefly illustrate the differences between thread state transitions and process state transitions. The states and state transitions of the thread model have different semantics than those of the process model as described below.

Thread state transitions. A thread can be in one of the following states:

- a *new* state; i.e., being created,
- a *running* state; its instructions are being executed,
- a *ready* state; the thread of control is waiting to be assigned to a processor,
- a *waiting* state; the thread of control is waiting or blocked for some signal or event to occur,
- a *terminated* state; the thread of control has finished execution.

The state transition diagram in figure 1 shows a common state transition model for threads [2].
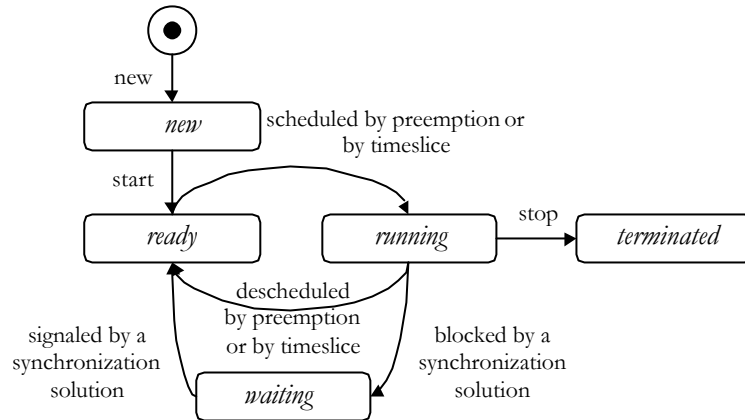
Figure 1, A thread state transition diagram [2].

Process state transitions. A process can be in one of the following states:

- an *instantiated* state; the process object is has been created or has successfully terminated, i.e., no thread is assigned to the process,

- a *running* state; the thread of control has been assigned to a process and the process has become active,

- a *preempted* state; the process has been preempted by some process running at a higher priority. The process is ready to be scheduled, but is not running,

- a *waiting* state; the thread of control is inactive or blocked, and is waiting to be notified,

- a *garbage* state; the process is terminated and will never be assigned to any thread; the process memory can be freed, for instance, by a garbage collector .

The state transition diagram in figure 2 shows the transitions between these process states.
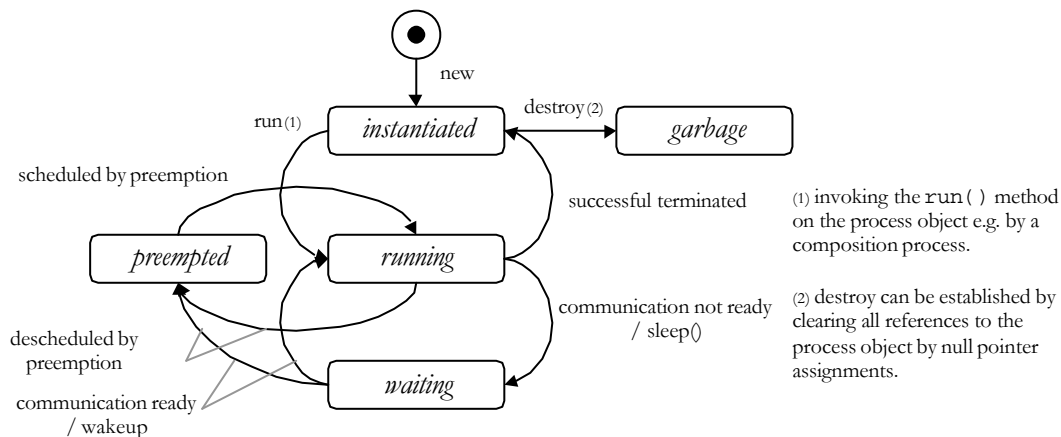

Figure 2, a process state transition diagram

The most important difference between the two diagrams is that more than one process may be *running* in parallel, but for each processor only one thread is *running* at any one time. In other words, a process state transition diagram applies to each process and is independent of the number of processors, while the thread state transition diagram only applies to one processor. However, the complete thread state transition diagram does underlie the process state transition diagram if these processes run on one processor. Thus, the user who works with processes need only understand the process state transitions for individual processes. This makes it easier to

analyse a design of a concurrent application. This treatment of debugging also holds down to coding and testing.

The process state transition diagram incorporates the notion of priority and distinguishes between preemptive and non-preemptive scheduling. A process that is preempted by a higher priority process must temporarily block and becomes *preempted*. A *preempted* process becomes *running* when no other higher priority process is *running*. Notice that timesharing of the system between processes is not shown in the process state transition diagram, because timesharing of processes is not of any concern to the processes, whereas the thread state transition diagram specifies a timeslice transition between the *running* and *ready* states. The process state transition diagram applies for software compilation and for hardware compilation, whereas, the thread state transition model only applies for software compilation. In a hardware compilation, the notion of pre-emption vanishes and the *preempt* state together with its transitions can be omitted. The rest of the states and transition does not change. The simple state transition that the process model offers is simple and clean, which makes it for the engineer easier to design and analyse a concurrent application correctly.

3 Communicating Processes in Java

This section describes the CSP process interface and CSP channel interface, as implemented in the CTJ package. It should be noted that all CTJ channels are valid CSP channels.

3.1 The CTJ process interface

Processes running in parallel do not need to know about each other. A process need only know about its own channels.

In Java, a parent process creates its child process and starts it executing by invoking its run method. Therefore, the *process interface* of a CTJ process consists of a `run()` method, and a set of input and output channels that are used by the process in order to communicate with other parallel processes.
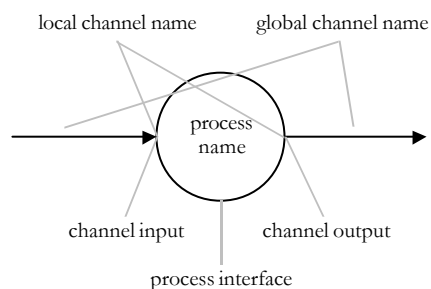
Figure 3, Graphical representation of active process interface.

3.2 The CTJ channel interface

In CTJ, a *channel interface* is the simplest communication interface; it might be thought of as a language primitive. This interface contains a *channel-input interface* that specifies a `read()` method and a *channel-output interface* that specifies a `write()` method. Processes communicate with other processes by reading or writing on channels that they both share by invoking these `read()` or `write()` methods. Special about CTJ channels is that they allow multiple readers and writers respectively reading and writing on channels simultaneously.

CTJ channels provide both a hardware-independent and a hardware-dependent framework. These frameworks are connected by the simple channel interface. In other words, channels map the software onto the hardware as illustrated in figure 4.

Hardware independence. Communication via channels provides a platform independent framework under which processes may be located either on the same processor, or distributed onto multiple processors. Processes *never* access hardware directly, and may only communicate with their environment via channels. As a result, an individual process does not know which processes are at the other end of its channels, nor what kind of hardware connects them.

Hardware dependence. Channels may actually establish a connection (a link) between two or more processors. In this case, special *link driver* objects are plugged into the channel using an alternative channel constructor. Link drivers control

Figure 4, Channel framework.

the underlying hardware and are the hardware dependent code in an application. The CTJ link driver framework is abstractly defined and can be extended as needed without affecting the system design or the hardware independent framework. (The link driver framework also provides special methods for interrupt handling. For more information about the link driver framework, see [11].)

Processes that do not use or create link drivers are fully hardware independent. Those processes that use link drivers are hardware dependent. Hardware dependent processes are considered to be *network builders*, as these can be used to setup and configure a network of processes and channels in order to map the software on the topology of the hardware.
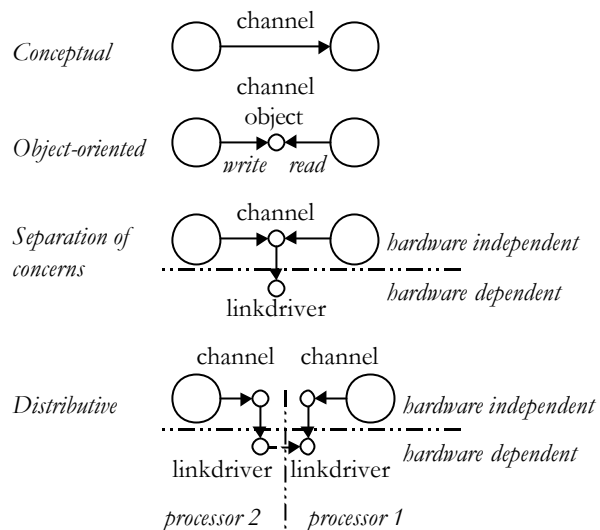
4   Concurrent Programming with CTJ

This section describes how to create Java applications using the communicating processes and composition constructs that are provided with the CTJ package.

4.1   Creating processes

A process is specified using the `csp.lang.Process` interface, which specifies a public `run()` method, as shown in listing 1. This is similar to Java's `java.lang.Runnable` interface.

```
public interface csp.lang.Process {
   public void run();
}
```
   Listing 1, the passive process interface class.

A process class must implement the `csp.lang.Process` interface and must provide a `run()` method implementation. Its `run()` method implements the sequential task that the process will perform when this method is invoked by another process. Listing 2 shows an example of this.

```
class MyProcess implements csp.lang.Process {

  // local declarations

  public MyProcess(channels and parameters) {
```

```
    // construct process
    }
    public void run() {          // do something
    }
}
```

Listing 2, example process class.

The constructor of the process specifies the process interface of channel inputs and outputs, and additional parameters for initiating the process state. When a process is instantiated, its constructor must set up all of the initial resources, such as input channels, output channels and parameters, before the `run()` method is called.

The `run()` method is the only public method that one process may directly invoke on another process. The `run()` method may implement real-time activities and is allowed to begin execution as long as sufficient resources are available to enable it to run reliably.

A process is an passive object before its `run()` method has been invoked. A process is an active object when its `run()` method has been invoked by some thread of control and has not yet completed execution. Therefore, a parent process should only invoke a child process's `run()` method when that child process is in passive state. Sharing a process by two or more processes is forbidden (design rule) and, therefore, its `run()` method can never be invoked simultaneously by multiple processes. This simple rule avoids race hazards and strictly separates each thread of control to enable a secure multithreading environment.

4.2   Producer/consumer example

The basis of creating processes is illustrated in the producer/consumer example below. Figure 3 shows the data-flow diagram (DFD) for two communicating processes, a producer process and a consumer process. The producer has some data available for the consumer. The data will be sent when both, the producer and consumer process are ready to communicate. The `Producer` class specifies a process interface with an output channel and the `Consumer` class specifies a process interface with an input channel.

The basic templates of the `Producer` and `Consumer` classes are defined as follows:

```
import csp.lang.*;

class Producer implements csp.lang.Process {

    ChannelOutput channel;

    public Producer(ChannelOutput out) {
        channel = out;
    }
    public void run() {
        // . . .
        //channel.write(object);
        // . . .
    }
}
```
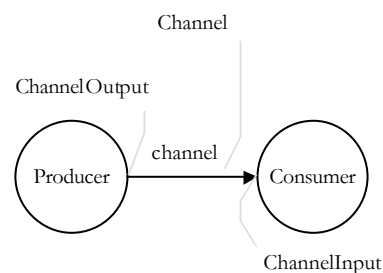

Figure 5, Conceptual DFD for producer consumer communication

```
class Consumer implements csp.lang.Process {

  ChannelInput channel;

  public Consumer(ChannelInput in) {
     channel = in;
  }
public void run() {
    // . . .
    //channel.read(object);
    // . . .
  }
}
```
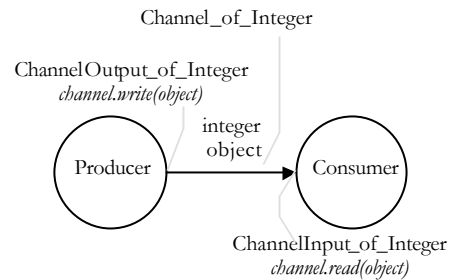
Listing 3, Producer/Consumer example.

The `ChannelInput` and `ChannelOutput` types are general interfaces that specify whether the channel will be used for input or output. However, no

Figure 6, Detailed producer/consumer example (DFD).

`read()` or `write()` methods are specified at this level, as they are type-less, as shown in Figure 5. The `ChannelInput` interface specifies the root interface type for all channel input interfaces. The `ChannelOutput` interface specifies the root interface type for all channel output interfaces. Message types (channel protocols) are provided by subclasses that specify compatible input and output methods for the channel. Specifying the type of message that will be communicated over a channel provides consistency and eliminates a source of errors. The "message-typed" channel classes define the actual `read()` and `write()` methods. Figure 6 and Listing 4 illustrate the use of a message-typed channel for `Integer` messages.

Suppose that the producer, described above, is producing integer values needed by the consumer. Then the producer and consumer processes can be defined as:

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

class Producer implements Process {

  ChannelOutput_of_Integer channel;
  Integer object;

  public Producer(ChannelOutput_of_Integer out) {
    channel = out;
    object  = new Integer();  // pre-allocate object
  }

  public void run() {
    // . . .
    object.value = 100;
    channel.write(object);
    // . . .
  }
}
class Consumer implements Process {

  ChannelInput_of_Integer channel;
  Integer object;

  public Consumer(ChannelInput_of_Integer in) {
    channel = in;
    object = new Integer();   // pre-allocate object
  }

  public void run() {
    // . . .
    channel.read(object);
    System.out.println(object);
```

```
    // . . .
  }
}
```
Listing 4, producer/consumer process classes.

The `ChannelOutput_of_Integer` interface extends `ChannelOutput,` to specifying the `write(Integer  object)` method, and extends `ChannelInput,` specifying the `read(Integer  object)` method. There is a one-on-one mapping between the graphical process interfaces (DFDs) and the textual process interfaces in the source code.

Both processes may be run in parallel using the parallel composition as shown in Listing 5. (See Appendix A for a complete listing of the files needed to run the producer/consumer problem. A distributed version is shown in Appendix B.)

```
Public static void main(String[] args) {
  Channel channel = new Channel();              // create channel object
  Process par = new Parallel(new Process[] {    // create parallel construct
    new Producer(channel),
    new Consumer(channel)
  });

par.run();                                      // run parallel composition
}
```
Listing 5, Producer/Consumer parallel composition example.

The `main()` method's thread invokes `par.run()` and then waits until both the consumer and producer processes have successfully terminated before continuing its execution.

4.3   CTJ Message and channel types

When two cooperating processes are both ready to communicate, the `value` of the `Integer` object will be *copied* from the producer process's space into the consumer process's space. Each process has a local `csp.lang.Integer` object. The `csp.lang.Integer` class, Listing 6, specifies a public `int value` attribute and is therefore different from the value in the `java.lang.Integer` wrapper, whose value is read-only.

```
package csp.lang;

class Integer implements java.io.Serializable {
  public int value;
  public Integer(int value) { this.value = value; }
  public String toString() { return String.valueOf(value); }
}
```
Listing 6, the Integer object class.

Types of messages that may be communicated are not limited to Integer. There are many default wrappers and channels implemented in the CTJ package for the Java standard data types. These are listed in Table 1.

csp.lang.…	Channel_of_…	ChannelInput_of_ …	ChannelOutput_of_ …
Boolean	Channel_of_Boolean	channel.read(csp.lang.Boolean)	Channel.write(csp.lang.Boolean)
Byte	Channel_of_Byte	channel.read(csp.lang.Byte)	Channel.write(csp.lang.Byte)
Char	Channel_of_Character	channel.read(csp.lang.Character)	Channel.write(csp.lang.Character)
Double	Channel_of_Double	channel.read(csp.lang.Double)	Channel.write(csp.lang.Double)
Float	Channel_of_Float	channel.read(csp.lang.Float)	Channel.write(csp.lang.Float)
Integer	Channel_of_Integer	channel.read(csp.lang.Integer)	channel.write(csp.lang.Integer)
Long	Channel_of_Long	channel.read(csp.lang.Long)	channel.write(csp.lang.Long)
Short	Channel_of_Short	channel.read(csp.lang.Short)	channel.write(csp.lang.Short)
Reference	Channel_of_Reference	channel.read(csp.lang.Reference)	channel.write(csp.lang.Reference)
Object	Channel_of_Object	channel.read(java.lang.Object)	channel.write(java.lang.Object)
Any	Channel_of_Any	channel.read()	channel.write()

Table 1, CTJ wrappers and channel interfaces.

5   Composition of processes in CTJ

A process begins to execute when its `run()` method is invoked. The invoking process's execution is then suspended until that `run()` method returns successfully.

CSP describes common compositions of processes, i.e., it specifies that processes can execute *in sequence*, *in parallel* or *by choice*. CTJ's `Sequential`, `Parallel`, `PriParallel`, `Alternative` and `PriAlternative` composition constructs are described in this section. These constructs are processes themselves that allow nesting of composition constructs. They automatically invoke the `run()` methods of their subprocesses using either the same or a separate thread of control, depending on their semantics. Once a network of processes has begun executing, these processes will be scheduled depending on channel communication and composition behavior.

The following sections describe building compositions of processes.

5.1   `Sequential`, the sequential composition construct

The CTJ package provides a special process that executes a list of processes in a sequential order. We call this process the sequential composition construct.

A Sequential composition construct is created using the `Sequential` class. The `Sequential` object itself is a process and is created as shown:

```
Sequential seq = new Sequential(Process[] processes);
```

The argument `processes` is an array of processes that begins executing when it's the Sequential construct's `run()` method is invoked, e.g.,

```
seq.run();
```

When the `run()` method of a sequential composition construct is invoked then its subprocesses are executed one at a time and in order. The Sequential construct process finishes when all of its subprocesses have finished.

The following example shows a sequential composition of three processes.

```
Sequential seq = new Sequential(new Process[] {
   new Process1(channel interfaces),
   new Process2(channel interfaces),
   new Process3(channel interfaces)
});

seq.run();
```

In this case, `Process1` will execute to completion first, followed by `Process2` and then by `Process3`. The `seq` process finishes successfully when all Process3 successfully finishes, i.e., when all three processes have successfully finished running in order

New processes can be added at the end of the process list at run-time, using

```
seq.add(new Process4(..));
```

or by adding multiple processes at a time

```
seq.add(new Process[] { new Process4(..), new Process5(..) });
```

A new process can be inserted at run-time, using

```
seq.insert(process, index);
```

Process `process` will be inserted at `index` of the process list.
A process can be removed from the process list, by using

```
seq.remove(process);
```

WARNING: The `add()`, `insert()`, and `remove()` methods may only be used outside of the `Sequential` construct. Furthermore, only the parent process of the `Sequential` construct may invoke these methods and only when the construct is not executing, i.e., the construct process must be in the *instantiated* state. This warning also applies for the constructs in the next sections. These restrictions provide a safe and reliable way to use dynamic constructs.

5.2 `Parallel`, the parallel composition construct

Another special process that CTJ provides is the parallel composition construct, which executes a list of processes in parallel. The `Parallel` class creates the `Parallel` composition construct. The `Parallel` object itself is a process and can be used in other constructs illustrated in section 5.6.

```
Parallel par = new Parallel(Process[] processes);
```

The argument `processes` is an array of processes that begin executing when the `run()` method of the `Parallel` construct is invoked, e.g.,

```
par.run();
```

The subprocesses of the `Parallel` composition construct execute in parallel. The construct finishes execution when all subprocesses have completed execution.

The following example shows a `Parallel` composition of three processes.

```
Parallel par = new Parallel(new Process[] {
   new Process1(channel interfaces),
   new Process2(channel interfaces),
   new Process3(channel interfaces)
});
```

```
par.run();
```

The processes `Process1`, `Process2`, and `Process3` will be executed in parallel, using separate threads of control for each. These will inherit the same priority as the `Parallel` process. The `par` process finishes successfully when *all* three of its subprocesses have successfully finished. New processes may be added at run-time, using

```
par.add(new Process4(..));
```

or by adding multiple processes at a time:

```
par.add(new Process[] { new Process4(..), new Process5(..) });
```

A process may be removed from the process list, using

```
par.remove(process);
```

5.3   `PriParallel,` the priority based parallel composition construct

CTJ also supports priorities. The `PriParallel` composition extends the `Parallel` composition to include priorities. Each process of the `PriParallel` construct is assigned a priority in successive order. The first process in the `PriParallel` process list is given the highest priority and the last process in the process list is given the lowest priority within the `PriParallel` construct. Currently, the maximum number of priorities per `PriParallel` is 8; where 7 are for user defined processes and one is reserved for an idle task, skip task, or garbage collector task. The reserved priority is private to the `PriParallel`. The restriction to 8 priorities allows quick priority sorting with the efficiency of order $O(2)$, i.e., a process can be placed into the correct priority queue in a maximum of two steps. PriParallel compositions may be nested within other `PriParallel` processes arbitrarily that enables the support of more than 7 priorities. This will be expanded upon later.

A process does not have a priority by itself; in other words, the user cannot assign a priority number to a process instead he or she may add the processes to a `PriParallel` construct. The philosophy behind this is that the priority number of a process is an implementation issue and not a design issue. The designer wants to specify a process that must be executed with a higher, equal, or lower priority than another process, rather than using some number. At the implementation level, the `PriParallel` process will solve the ordering of the priorities.

The `PriParallel` class creates the priority based parallel composition construct as shown here:

```
PriParallel pripar = new PriParallel(Process[] processes);
```

The argument `processes` is an array of processes that begin executing when the `run()` method of the `PriParallel` construct is invoked, e.g.,

```
pripar.run();
```

The following example shows a prioritized parallel composition of three processes.

```
PriParallel pripar = new PriParallel(new Process[] {
  new Process1(channel interfaces),              // priority 0
  new Process2(channel interfaces),              // priority 1
  new Process3(channel interfaces)               // priority 2
});

pripar.run();
```

The processes `Process1`, `Process2`, and `Process3` will be executed in parallel with successively lower priorities. Process `Process1` (at index 0) has the highest priority. All processes in a process list with index 6 and higher would share the lowest priority in this construct. The `pripar` process finishes successfully when all three processes have successfully finished.

Increasing the maximum number of priorities, i.e., more than 7, is possible by nesting a `PriParallel` process within a `PriParallel` process. The following example illustrates a `PriParallel` construct with 49, i.e., $=7^2$, priorities.

```
PriParallel pripar = new PriParallel(new Process[] {
  new PriParallel(new Process[] {               // priority 0
    Process1_1(channel interfaces),             // priority 0.1
    Process1_2(channel interfaces)              // priority 0.2
    Process1_3(channel interfaces)              // priority 0.3
    Process1_4(channel interfaces)              // priority 0.4
    Process1_5(channel interfaces)              // priority 0.5
    Process1_6(channel interfaces)              // priority 0.6
    Process1_7(channel interfaces)              // priority 0.7
  }),
  new PriParallel(new Process[] {               // priority 1
    Process2_1(channel interfaces),             // priority 1.1
    ..as before..                               // priority 1.2-6
    Process2_7(channel interfaces)              // priority 1.7
  }),
  new PriParallel(new Process[] {..as before..}),  // priority 2.1-2.7
  new PriParallel(new Process[] {..as before..}),  // priority 3.1-3.7
  new PriParallel(new Process[] {..as before..}),  // priority 4.1-4.7
  new PriParallel(new Process[] {..as before..}),  // priority 5.1-5.7
  new PriParallel(new Process[] {..as before..})   // priority 6.1-6.7
});

pripar.run();
```

New processes may be added at run-time, using

```
pripar.add(new Process4(..));
```

or by adding multiple processes at a time:

```
pripar.add(new Process[] { new Process4(..), new Process5(..) });
```

A new process may be inserted at run-time using

```
pripar.insert(process, index);
```

Process `process` will be inserted at `index` of the process list.
A process may be removed from the process list, using

```
  pripar.remove(process);
```

The order of priorities will automatically be applied to the new process list.

5.4   `Alternative,` the alternative composition construct

Sometimes we need to choose one process out of a set of processes that are simultaneously committed in communication. An `IF-THEN-ELSE` construct works for Boolean expressions but not for communication events. A communication event does not return `true` or `false`. Therefore, a special process, called the alternative composition construct, is provided that implements the CSPs choice operator. The alternative composition construct combines a number of processes guarded by inputs, outputs and timeouts. The alternation performs the process associated with a guard, which is ready [9]. If no guard is ready the alternation will suspend until a guard becomes ready. A suspended alternative construct consumes no time. As soon as one guard becomes ready it will resume the alternative construct followed by the execution of the process it guarded. When the selected process finishes, the execution of the `Alternative` construct finishes as well.

The `Alternative` class defines the `Alternative` composition construct. The `Alternative` object itself is also a process instantiated by:

```
Alternative alt = new Alternative(Guard[] guards);
```

The argument `guards` is an array of `Guard` objects. A `Guard` object is an instance of the `Guard` class. There are two ways one can use the `Alternative` in CTJ: with a select-based method or as a compositional construct. We will start with explaining the compositional approach that is almost similar as the sequential and parallel constructs as described in the previous sections.

The compositional `Alternative` construct starts by invoking its `run()` method, e.g.,

```
alt.run();
```

The following example shows an `Alternative` composition for three guarded processes.

```
Alternative alt = new Alternative(new Guard[] {
   new Guard(channel1, new Process1(channel1, ..)),
   new Guard(channel2, new Process2(channel2, ..)),
   new Guard(channel3, new Process3(channel3, ..))
});
alt.run();
```

Here, $channel_i$ is an input channel or output channel of $Process_i$. The `Guard` with $Process_i$ is ready when $channel_i$ has a message waiting to be read and becomes a candidate for selection. The `alt` process waits until at least one guard becomes ready and completes successfully when one of the ready guards is selected and its respective process has successfully executed. If more than one `Guard` is ready when the `Alternative` is executed, then one `Guard` will be 'randomly' selected; theoretically, a non-deterministic choice. CTJ's `Alternative` construct makes its selections *fairly*, i.e., when more than one guard is ready, the guard to execute will be selected according to a first-come-first-served queuing mechanism and the process that it guards will then be executed.

New guards may be added at run-time, using

```
alt.add(new Guard(channel4, new Process4(channel4, ..));
```

or by adding multiple guards at a time:

```
alt.add(new Guard[] {
   new Guard(channel4, new Process4(channel4, ..),
```

15

```
            new Guard(channel5, new Process5(channel5, ..)
});
```

A guard may be removed from the guard list, using

```
alt.remove(guard);
```

Processes that are specified in a guard can also be written as anonymous processes as shown next.

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

    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++) {
      alt.run();                   // make the selection and run the response
    }                              // 20 times
```

The select-based `Alternative` construct starts by invoking the `select()` method, e.g.,

```
    i = alt.select();
```

The index `i` specifies the guard that was selected. This method does not execute any specified process of the selected guard. An example is given below.

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

    Integer n = new Integer();

    for (int i=0; i<20; i++) {
      int index = alt.select();               // wait for a channel
      inChannel[index].read(n);               // read from selected channel

      … do something with n
    }
```

A new modification in CTJ (since revision 17) is that every object that inherits the `Guard` class can play the role of a guard. In CTJ a channel can also play the role of a guard, which simplifies the code. This can only be used with the select-based `Alternative` construct.

```
    Alternative alt = new Alternative(new Channel[] { // the Alternative
                    inChannel[0],                // can only be used
                    inChannel[1]                 // with its select()
                });                              // method
    Integer n = new Integer();

    for (int i=0; i<20; i++) {
      int index = alt.select();                  // wait for a channel
```

```
    inChannel[index].read(n);                    // read from selected channel
  … do something with n
}
```

5.4.1   Unconditional and conditional guards

The `Guard` object guards a process by signaling the `Alternative` construct when the first occurrence of a communication input event for the channel is ready. As shown in the previous section, a `Guard` object may be declared as follows,

```
Guard guard = new Guard(channel, new Process(channel,..));
```

The `Guard` becomes `true` when argument `channel` (input or output interface) is ready and has data available to be read by `Process`. The `Guard` described above always participates in the `Alternative` construct and is called an *unconditional* guard. A guard may also be *conditional,* i.e., the `Guard` is enabled and participates in the `Alternative` construct only if some condition is true; otherwise, the `Guard` is disabled and omitted by the `Alternative` construct, in which case its process will never be selected. For example,

```
Boolean condition = new Boolean();
condition.value = true;
Guard guard = new Guard(condition, channel, new Process(channel,..));
```

If `condition.value` is `true` the guard will check `channel`, otherwise the `Guard` is omitted and the process will not be selected. Any part of the alternative construct including its guarded processes may update variable `condition.value` at any time. On the basis of these conditions one can implement its state transitions in a simple, safe and elegant manner.

A conditional `Guard` declared using

```
new Guard(new Boolean(true), channel, new Process(channel,..))
```

is equivalent to

```
new Guard(channel, new Process(channel,..)).
```

Applying conditional guards is useful for implementing state transitions.

The `Guard` itself is not a process, but rather a passive object that guards a process. The `Alternative` object will check all of the guards for readiness, and will wait until at least one guard becomes ready, i.e., channel has input or output. At that time, the process belonging to a ready guard will be selected and executed and must read the message on `channel`.

See the `Guard` class documentation for all possible guards.

5.4.2   Skip-guards

In circumstances where the `Alternative` construct should continue when no channel is ready then a skip-guard provides this behavior. A skip-guard is a guard that does not wait for an event to be ready. Skip-guards are,

Unconditional skip-guards

`Guard()`	is always true and performs a skip if selected
`Guard(process)`	is always true and performs the process if selected

Conditional skip-guards (see section 5.4.1)

`Guard(condition)`	performs skip if selected
`Guard(condition, process)`	performs process if selected

CTJ provides a special process Skip that can play the role of a process or the role of a guard, as in,

`Guard(new Skip())`	is always true and performs a skip if selected (same as `Guard()` in the role of a guard)
`Skip()`	is always true and performs a skip if selected (same as `Guard(new Skip())` in the role of a guard)

The conditional versions (see section 5.4.1) are,

`Guard(condition, new Skip())`	performs a skip if selected
`Skip(condition)`	performs a skip if selected

5.4.3 Timeout-guards

A process can also be guarded by a timeout event. A timeout-guard can be specified by

Unconditional timeout-guards

`Guard(time)`	becomes ready after the specified time (long) and performs a skip is selected
`Guard(time, process)`	becomes ready after the specified time and executes the specified process if selected

Conditional timeout-guards (see section 5.4.1)

`Guard(condition, time)`	performs a skip after the specified time and if selected
`Guard(condition, time, process)`	performs the specified process after the specified time and if selected

For readability, a `Timeout` class is included that defines a timeout-guard that is the similar as the `Guard(.., long time, ..)` guards above.

`Timeout(time)`	same as `Guard(time)`
`Timeout(time, process)`	same as `Guard(time, process)`
`Timeout(condition, time)`	same as `Guard(condition, time)`

`Timeout(condition, time, process)` same as `Guard(condition, time, process)`

When multiple timeout-guards in the guard list of the `Alternative` construct are specified then the guard with the smallest timeout will be active. When there are multiple timeout-guards with the smallest and equal timeout then one will be selected. One timeout-guard per `Alternative` is recommended.

5.5 `PriAlternative`, the priority based `Alternative` composition construct

The `PriAlternative` class creates the priority based `Alternative` composition construct. The `PriAlternative` class extends the `Alternative` class and overrides the 'fair' choice mechanism with an 'unfair' or prioritized based choice mechanism. The `PriAlternative` object itself is also a process. The `PriAlternative` construct is similar to the `Alternative` construct. It is instantiated by

```
PriAlternative prialt = new PriAlternative(Guard[] guards);
```

The argument `guards` is an array of guard objects. A `Guard` object is an instance of the `Guard` class.

The `PriAlternative` construct starts by invoking the `run()` method.

```
prialt.run();
```

The following example shows a `PriAlternative` composition for three processes.

```
PriAlternative prialt = new PriAlternative(new Guard[] {
   new Guard(channel1, new Process1(channel1, ..)),
   new Guard(channel2, new Process2(channel2, ..)),
   new Guard(channel3, new Process3(channel3, ..))
});

prialt.run();
```

The `prialt` process waits until at least one guard becomes ready and finishes successfully when one of the three processes is selected and has successfully executed. The `Guard` with `Process`$_i$ may be selected when `channel`$_i$ is ready. Here, `channel`$_i$ is an input channel or output channel of `Process`$_i$. If more than one guard is ready than the guard with the lowest index will be selected and the process of the selected guard will be executed.

New guards may be added at run-time, using

```
prialt.add(new Guard(channel4, new Process4(channel4, ..));
```

or by adding multiple guards at a time:

```
prialt.add(new Guard[] {

   new Guard(channel4, new Process4(channel4, ..),
   new Guard(channel5, new Process5(channel5, ..)
});
```

A new `Guard` may be inserted at run-time, using

```
prialt.insert(guard, index);
```

Guard guard will be inserted at index in the guard list. The priority of guard and other guards below it will automatically be incremented, i.e., a higher number indicates a lower priority.

A process may be removed from the process list, using

```
prialt.remove(guard);
```

The priority of the guards below guard will automatically be decremented.

5.6   Nested composition constructs

The Sequential, Parallel, PriParallel, Alternative and PriAlternative composition processes may be nested within other composition processes. For example, two sequential constructs may be executed in parallel:

```
Process process = new Parallel(new Process[] {
   new Sequential(new Process[] {

      new Process1(channel interfaces),
      new Process2(channel interfaces)
   }),

   new Sequential(new Process[] {
       new Process3(channel interfaces),
      new Process4(channel interfaces)
   })
});

process.run();
```

Or, two parallel constructs may be executed in sequence:

```
Process process = new Sequential(new Process[] {
   new Parallel(new Process[] {

      new Process1(channel interfaces),
      new Process2(channel interfaces)
   }),
   new Parallel(new Process[] {
      new Process3(channel interfaces),
      new Process4(channel interfaces)
   })
});

process.run();
```

Alternative constructs may be nested in the same way. Special is that Alternative constructs can play the role of a process or the role of a guard, as illustrated below

```
Process process = new Sequential(new Process[] {
   new Parallel(new Process[] {
      new Process1(..),
      new Process2(..)
   }),

   new Alternative(new Guard[] {
      new Guard(true, channel1, new Process3(channel1, ..)),
      new Guard(false, channel2, new Process4(channel2, ..))
      new PriAlternative(new Guard[]
```

```
        new Guard(channel4,
          new Sequential(new Process[] {
            new Process5(channel4, ..),
            new Process6(..)
          })),
        new Guard(channel5,
          new Sequential(new Process[] {
            new Process7(channel5, ..),
            new Process8(..)
          })),
    }),

    new Parallel(new Process[] {
      new Process9(..),
      new Process10(..)
    })
  })
});

process.run();
```

6 Conclusions

The *Communicating Threads for Java* (CTJ) package is an implementation of the CSP model that results in much simpler and more reliable thread constructs than those provided using the Java thread model. The CTJ package provides a small set of design patterns, based on processes, compositions and channels, which are sufficient for doing concurrent programming in Java. An important advantage of CTJ is that the designer or programmer can apply a full set of rules or guidelines for eliminating race hazards, deadlock, livelock, and starvation during design and implementation phases.

Reasoning about the behavior of processes becomes more abstract and easier for developers, because the behavioral semantics of process synchronization and scheduling is simplified through channel communication and compositional constructs. As a result, tracing process states and debugging processes is easier than when using threads directly. The concept of processes, channels and link drivers specifies a logical separation of hardware dependent and hardware independent sections. This framework is expected to pay off when Java software is developed for real-time and embedded systems.

7 References

[1] B. Lewis and D.J. Berg, *A guide to multithreading programming: Threads Primer*, Prentice Hall, USA, 1996.

[2] A. Silberschatz and P. Galvin, *Operating Systems Concepts*, Addison-Wesley Publishing, Fourth Edition, 1994.

[3] D. Lea, *Concurrent Programming in Java*, Addison-Wesley, Massachusetts, USA, 1996.

[4] POSIX committee on multithreading standards 1003.1c, publications at http://www.nist.gov/ and http://standards.ieee.org/.

[5] K. Arnold and J.A. Gosling, *The Java Programming Language*, Addison-Wesley, Massachusetts, USA, 1996.

[6] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, F. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, N.J., USA, 1991.

[7] G. Booch, J. Rumbaugh and I. Jacobson, *Unified Modeling Language User Guide*, Addison-Wesley, Massachusetts, USA, 1998.

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

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

[10] J. Davies and S. Schneider, *Real-Time CSP*, UK, 1995.

[11] University of Twente, http://www.rt.el.utwente.nl/javapp.

[12] G. Jones, On Guards, *Parallel Programming of Transputer Based Machines*, IOS Press, 1987

[13] R. Grebe et al., *A Flexible High Speed Distributed Control System for Aircraft Testing*, Transputer Applications and Systems '93, IOS Press, 1993.

8   Appendix A

The set of files shown in this appendix represents all of the files need to execute the Producer/Consumer program discussed in Section 4.2.

8.1   Producer.java

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

class Producer implements Process {

  ChannelOutput_of_Integer channel;
  Integer object;

  public Producer(ChannelOutput_of_Integer out){
    channel = out;
    object  = new Integer(); // pre-allocate object
  }

  public void run() {
    object.value = 100;
    channel.write(object);
  }

}
```

8.2   Consumer.java

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

class Consumer implements Process {
  ChannelInput_of_Integer channel;
  Integer object;

  public Consumer(ChannelInput_of_Integer in) {
    channel = in;
    object = new Integer();   // pre-allocate object
  }

  public void run() {
      channel.read(object);
      //System.out.println(object); must be replaced by either
      //java.lang.System.out.println(object); or
      //csp.lang.System.out.println(object);
      //otherwise get ambiguous class statement
      java.lang.System.out.println(object);
  }
}
```

8.3   PCMain.java

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

public class PCMain {

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

```
  public 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 Producer(channel),
                            new Consumer(channel)
                        });

    // run parallel composition
    par.run();

  }

}
```

9 Appendix B

The following files are needed in addition to Producer**.**java and Consumer.java from Appendix A to distribute the program.

9.1   PC1Main.java

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

public class PC1Main {

  public static void main(String[] args) {
    new PC1Main();
  }

  public PC1Main() {

    // create channel object
    final Channel_of_Integer channel =
      new Channel_of_Integer(new TCPIP("", 1701, TCPIP.REALTIME));

    // create parallel construct with a list of processes
    Process par = new Parallel(new Process[] {
      new Producer(channel)
    });

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

9.2   PC2Main.java

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

public class PC2Main {

  public static void main(String[] args) {
    new PC2Main();
  }

  public PC2Main() {

    // create channel object
    final Channel_of_Integer channel =
      new Channel_of_Integer(new TCPIP(1701, TCPIP.REALTIME));

    // create parallel construct with a list of processes
    Process par = new Parallel(new Process[] {
      new Consumer(channel)
    });

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