Architectures, Languages and Techniques
B.M. Cook (Ed.)
IOS Press, 1999

Communicating Threads for Java™

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

Abstract. The Java* thread model provides support for multithreading within the language and runtime system of Java. The Java synchronization and scheduling strategy is poorly specified and turns out to be of unsatisfactory real-time performance. The idea of Java is to let the underlying operating system specify the synchronization and scheduling principles. This may possibly result in different behavior on different operating systems whereas Sun claims Java to be system independent – "write once, run everywhere". In this paper we present a comprehensive specification for a new thread model for the Java platform.

The theory of CSP fully specifies the behavior of synchronization and scheduling of threads at a higher level of abstraction, which is based on processes, compositions and synchronization primitives. The CSP concept is well thought-out and has been proven to be successful for realizing concurrent software for real-time and embedded systems. The *Communicating Threads for Java* (*CTJ*) packages that is presented in the paper provides a reliable CSP/thread model for Java. The CTJ software is available from our URL http://www.rt.el.utwente.nl/javapp.

1 Introduction

The core Java thread model is derived from traditional multithreading concepts. Literature about threads and thread synchronization [1,2] is mostly about understanding the operating system or low-level concepts of multithreading. Design patterns concerning concurrent programming with threads [3] are mostly about implementation issues instead of conceptual issues. Reasoning about threads in an application can be different for each program construct. Hence, such a construct is an implementation issue. Threads are less abstract and are a major cause of increasing complexity. The freedom that is given by the flexible thread application programming interfaces (API) [1,4,5] increases the potential dangers of race hazards, deadlock, livelock, starvation etc. The programmer must be very careful and should apply a diversity of rules, guidelines or design patterns.

Analyzing a multithreaded program by debugging thread states can be extremely difficult when using such an API. Theoretical methods are developed for analyzing the behavior of a multithreaded application and for eliminating race hazards, deadlocks, livelock, starvation etc.

* 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.

Unfortunately, the Java thread model entails a serious gap between theory and practice. Furthermore, it is difficult for the designer to apply multithreaded behavior to current design methods, such as OMT [6] and UML [7].

The theory *Communicating Sequential Processes* (CSP) [8, 9, 10] offers a mathematical notation for describing patterns of communication by algebraic expressions. It comprehends a formal proof for analyzing, verifying and eliminating race hazards, deadlocks, livelock, starvation etc. The theory describes a complete set of thread behavior patterns and synchronization primitives in terms of processes, compositions and channel communication. The *Communicating Threads for Java* (CTJ) package implements the CSP model (i.e. processes, compositions and channels) in Java, which comprises much simpler and more reliable thread patterns than the Java thread model. The CTJ package provides a small set of design patterns that is sufficient for concurrent programming in Java. An important advantage of CTJ is that the designer or programmer can apply to a rich set of rules or guidelines for eliminating race hazards, deadlock, livelock, starvation etc. during design and implementation.

Understanding the concept doesn't require extensive knowledge about the theory of CSP. This paper does not treat the mathematics of CSP, but treats the CSP model for Java. The CTJ package builds a bridge between theory and practice of CSP.

2 Java threads versus CSP processes

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. registers, instruction pointer and stack pointer). A process encapsulates its data and methods – in the same way as objects do – and 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.

A CSP process is an active object whose instructions are executed by one thread of control that is encapsulated within the process. Henceforth, a process is referred to a CSP process. A process thread does *never* overlap threads of other processes. Cooperation between processes is established by channel communication. Channels are passive and primitive objects between processes, on which processes can send and receive data messages. Updateable data should be kept local within the process memory space and does not need to be synchronized. Only read-only data may be shared between processes.

Processes can start processes but they *may not* directly control other processes except themselves. Besides safety, also debugging of processes is far less difficult than tracing thread behavior based on the Java thread model.

The behavior of Java threads is poorly specified and strongly depends on the behavior of the underlying operating system. Therefore, thread behavior may behave differently on different Java virtual machines (JVM), whereas process behavior should always behaves similar on any JVMs.

2.1 Synchronization primitives

In Java, more than one thread may be assigned to an 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 must control each thread by a diversity of methods, which must be used in a proper way. This set of methods (see java.lang.Thread class and java.lang.Object class in [5]) provides a basic and flexible multithreading concept. The `synchronized` clause (or monitor construct [5]) protects a critical region around the shared data that allows only one thread to enter the region at a time. Additionally, the `wait()/notify()` methods perform conditional queuing of threads within the critical region. A monitor synchronization construct involves keeping track of more than one method. However, this synchronization construct is expensive for just one single thread of control. It is not always trivial to determine if methods or regions must be synchronized. Certain design patterns [3] can solve this problem, but they can make your program more complex than needed. Moreover, to get synchronization between threads correctly can be difficult and error-prone.

Channels are special objects that perform communication between processes in a predefined way. Channels are passive intermediate objects between processes and take care of synchronization, scheduling, and the data transfer of the messages. Communication via channels is thread-safe, most reliable and very fast. Furthermore, the programmer will be freed from complicated synchronization and scheduling constructs. CSP channels are initially unbuffered and fully synchronized according to the *rendezvous principle*; the writer process waits until the reader process gets ready or the reader process waits until a writer process gets ready. The CTJ channel permits zero or more buffering in the channel, according to your requirements.

Thinking in terms of processes is more abstract and more cognitive for developers than thinking in terms of threads as will be discussed in the next section. The simplicity of using channels over the monitor construct is an important motivation for this statement.

2.2 State transitions

For each processor there will be only one thread running at a time. A multiprocessor system with *n* processors can run *n* threads simultaneously. However, a uniprocessor system can run multiple threads through scheduling [1,2]. As a thread or process executes, it changes state. The states and the state transitions of threads and processes do not have to be equal, although they run on the same system. We will briefly illustrate the differences between the thread state transitions and the process state transitions. The latter is introduced in this paper.

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

- a *new* state; a new thread is being created,
- a *running* state; 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 (STD) shown in figure 1 expresses a common state transition model for threads [2].

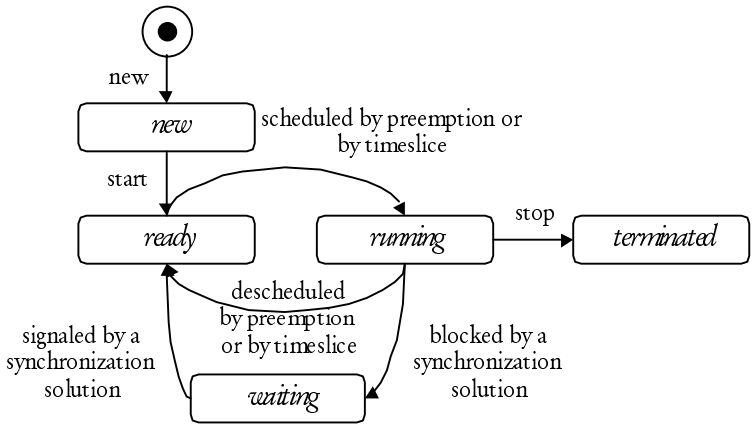

Figure 1: A thread state transition diagram (STD) after [2].

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

- an *instantiated* state; process object is being created or successfully terminated – 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 not running,
- a *waiting* state; the thread of control is inactive or blocked and waits to be notified,
- a *garbage* state; the process is terminated and will never be assigned to any thread – the Java garbage collector may cleanup the object.

The state transition diagram (STD) shown in figure 2 expresses the transitions between these states in which a process may be situated.

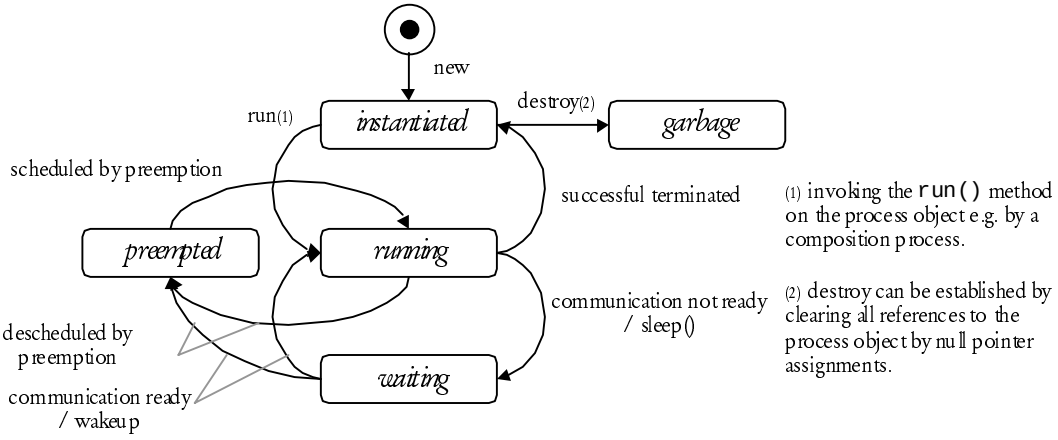

Figure 2: A process state transition diagram (DFD).

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 a time. In other words, a process state transition diagram applies to each process and is independent of the number of processors – more than one process may be running at a time. Whereas, an entire thread state transition diagram strongly depends on the number of processors. However, the entire thread state transition diagram lies underneath the process state transition diagram. Thus, the user who is concerned with processes only needs to understand the process state transitions for the individual process.

The process state transition diagram distinguishes preemptive and non-preemptive scheduling. A process that is preempted by a higher priority process must temporarily hold and becomes *preempted*. A *preempted* process becomes *running* when no other higher priority process is *running*. A process that must wait for some event to occur is just *waiting*. Timesharing of threads are hidden in the process state transition diagram, because timesharing of processes is not of any concern of processes.

3 Communicating Processes in Java

The CSP process interface and CSP channel interface, as implemented by the CTJ-package, are described in this section.

3.1 The process interface

Parallel running processes don't see each other. All each process sees are its channels and that's all it needs. In Java, a parent process creates its child process and starts it by invoking its `run()` method. Therefore, the *process interface* of the CTJ process is specified by a *passive process interface*, specifying the `run()` method, and an *active process interface*, specifying the set of channel inputs/outputs that is used by the process.

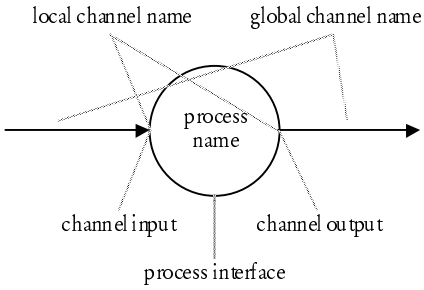

Figure 3: Graphical representation of active process interface.

The process interface can be derived from data-flow models, which are labeled and directed graphs consisting of process symbols (bubbles) and flow symbols (arrows), representing processes and channels. Processes are connected by flows and they specify the direction of the messages. Figure 3 illustrates a graphical representation of the active process interface of a process with one input and one output channel.

The process name encompasses the functional description of the process. The global channel names express the type of the message that will be passed between processes. The border of a process specifies its input and output of the process; incoming flows specify input channels on which the process can read and outgoing flows specify output channels on which the process can write. Input and output channels have local channel names inside the processes that may be different from the global channel names outside the processes, because they may exist in different contexts. A global channel is shared by more than one process on which they can

input and output. A local channel should be either input or output. One should be careful when a process can input *and* output on the same channel. The distinct channel input and output interfaces provide safety; one can only read on a channel input and one can only write on a channel output.

3.2 The channel interface

The CTJ *channel interface* is a primitive interface containing a *channel-input interface* that specifies a `read()` method, a *channel-output interface* that specifies a `write()` method, and a `getName()` method that returns the channel name. Processes communicate with other processes by reading or writing on shared channels by invoking the `read()` or `write()` methods. Also, the CTJ channel allows multiple readers and writers. Communication via channels, as built by the CTJ-library, provides a hardware independent and a hardware dependent framework. Both separated frameworks are connected by the simple interface of the channels. In other words, channels map the software onto the hardware as illustrated in figure 4.

Hardware independence. Communication via channels provide a platform independent framework where processes may be located on the same system or distributed across other systems. Processes *never* access hardware directly, but they may only communicate with their environment via channels. As a result, processes do not know what processes are at the other side of a channel and they do not know what hardware is in between. Processes may communicate with something that is virtually present.

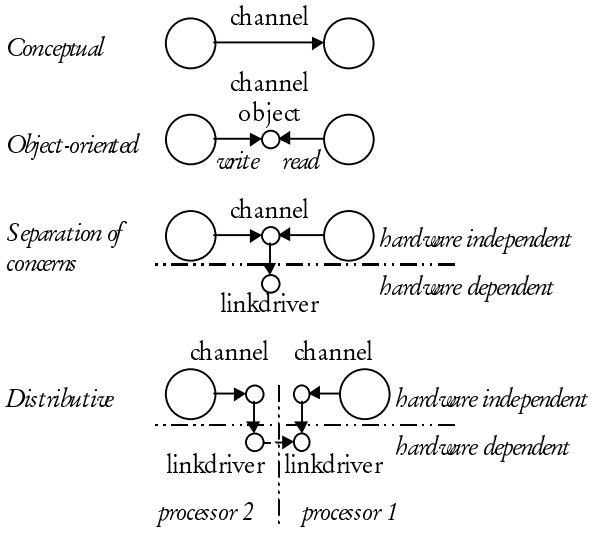

Figure 4: Channel framework.

Hardware dependence. Channels can establish a link between two or more systems via some hardware. Special *link driver* objects can be plugged into the channel. Link drivers control the hardware and should be the hardware dependent code in the application. The link driver framework is abstractly defined and can be extended as needed without influencing the design or the hardware independent framework. The link driver framework also provides interrupt handling. Information about the link driver framework can be found in [11].

All processes that do not create link drivers are fully hardware independent. Those processes that create link drivers are exceptional in that they are more or less hardware dependent, because link drivers directly represent hardware. These types of processes are considered to be *network builders*. Network builders setup and configure the network of processes and channels for the topology of the hardware.

4 Concurrent Programming with CTJ

This section describes how to create communicating processes and composition constructs in Java that are provided with he CTJ-package.

4.1 Create your process in Java

A process is defined by the `csp.lang.Process` interface. The `csp.lang.Process` interface defines a public `run()` method, see listing 1.

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

Listing 1, passive process interface class.

A process class must implement the `csp.lang.Process` interface. Note: The `csp.lang.Process` interface is similar to the `java.lang.Runnable` interface, but we prefer the name `Process`.

The `run()` method implements the runnable body of the process that will be invoked by another process and performs a sequential task.

```
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 name, channel input and channel output interfaces, and additional parameters for initiating the process state. The `run()` method is the only public method a process may invoke on another process, which is safe when this process was not running. No more than one process may invoke the `run()` method at a time. The `run()` method may implement real-time activities and is allowed to begin execution unless sufficient resources are available to enable it to run reliably. At instantiation of the process, the constructor must setup all resources, such as channel inputs/outputs and parameters, before the `run()` method is called.

4.2 Producer/consumer example

In the following producer/consumer example the basic layout of creating processes is illustrated. Figure 3 shows a data-flow diagram (DFD) two communicating processes, i.e. a producer process and a consumer process. The producer has some results available for the

consumer. The result will be send when both the producer and consumer process is ready to communicate. Producer specifies an active process interface with an output channel and consumer specifies an active process interface with an input channel.

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);
    // . . .
  }
}

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.

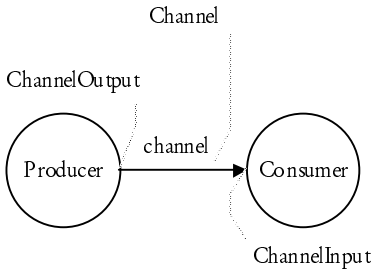

Figure 5: Conceptual producer/consumer representation (DFD).

The `ChannelInput` and `ChannelOutput` types are general interfaces, which distinguish between channel-input and channel-output. They do not specify any `read()` or `write()` methods here, because they are type-less (figure 5). Channel inputs and outputs must also specify the message type to provide compatible process interfaces. Message-typed channels provide consistency and eliminate a source of errors. The message-typed channel classes define the actual `read()` and `write()` methods. Figure 6 and listing 4 illustrates the use of a message-typed channel for `Integer` messages.

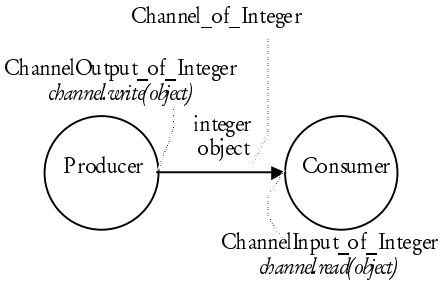

Figure 6: Detailed producer/consumer example (DFD).

Two processes communicating `Integer` objects can be defined as,

```
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);
    // . . .
  }
}

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.

Interface `ChannelOutput_of_Integer` extends `ChannelOutput` and specifies the `write(Integer object)` method. Interface `ChannelInput_of_Integer` extends `ChannelInput` and specifies the `read(Integer object)` method. There is a one-on-one mapping between the graphical process interface (DFDs) and the textual process interface (source code).

Both processes can run in parallel by the parallel composition that is given in listing 5.

```
public void main(String[] args)
{
  Channel channel = new Channel();        // create channel object
```

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

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

Listing 5, producer/consumer parallel composition example.

The thread in the `main()` method invokes `par.run()` and waits until the consumer and producer processes are successfully terminated.

4.3 Message and channel types

When the communication between two cooperating processes is ready, the `value` will be *copied* from the producer process space to the consumer process space. Each process keeps 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 `java.lang.Integer` wrapper, which 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.

There are many default wrappers and channels implemented in the CTJ package for the standard data types (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_Char	channel.read(csp.lang.Char)	Channel.write(csp.lang.Char)
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)
Proxy	Channel_of_Proxy	channel.read(csp.lang.Proxy)	channel.write(csp.lang.Proxy)
Object	Channel_of_Object	channel.read(java.lang.Object)	channel.write(java.lang.Object)
Any	Channel_of_Any	channel.read()	channel.write()

Table 1, wrappers and channel interfaces.

5 Composition of processes

Basically, processes execute when their `run()` method gets invoked. The caller process waits until the `run()` method returns successfully.

```
MyProcess process = new MyProcess();   // create process object (instantiated state)
process.run();                         // run process           (running state)
```

The theory of CSP describes common compositions on which processes are executed; i.e. processes can execute *in sequence*, *in parallel* or *by choice*. In this section the `Sequential`, `Parallel`, `PriParallel`, `Alternative` and `PriAlternative` composition constructs (respectively equivalent to composition operators in CSP: $;, ||, \overleftarrow{||}, \Box, \overleftarrow{\Box}$) are described. These special constructs are useful for building compositions of processes. They are processes themselves that allow nesting of composition constructs. They automatically invoke the `run()` methods on processes by the same or by a separate thread of control, according to their specific behavior.

Building compositions of processes with the basic composition constructs are described in the next sections. Once the network of processes is started, these processes will be scheduled on channel communication and composition behavior.

5.1 The sequential composition construct - Sequential

The sequential composition construct executes processes one at a time. The construct terminates when all processes have been terminated. The sequential composition construct is created by the `Sequential` class. The sequential object itself is a process.

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

The argument `processes` is an array of processes. The construct starts by invoking the `run()` method.

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

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();
```

`Process2` will run after `Process1` has successfully terminated. `Process3` will run after `Process2` has successfully terminated. The `seq` process terminates successfully when all processes are successfully terminated.

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

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

or by adding multiple processes at once

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

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

```
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

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

WARNING: The methods `add()`, `insert()`, and `remove()` may only be used outside the construct. Only the parent process of the construct may invoke these methods when the construct is not running – the construct process must be in the *instantiated* state. This restriction provides a safe and reliable way to use dynamic constructs. This warning also applies for the other constructs in the next sections.

5.2 The parallel composition construct - Parallel

The parallel composition construct executes processes in parallel. The construct terminates when all processes have been terminated. The `Parallel` class creates the parallel composition construct. The parallel object itself is a process.

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

The argument `processes` is an array of processes. The construct starts by invoking the `run()` method.

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

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. Each of them gets a separate thread of control with the same priority as the parallel process. The `par` process terminates successfully when *all* three processes are successfully terminated.

New processes can be added at run-time, by

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

or by adding multiple processes at once

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

A process can be removed from the process list, by

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

5.3 The priority based parallel composition construct - PriParallel

The priparallel composition extends the parallel composition with priorities. Each process of the priparallel construct will be given a priority in successive order, whereas each process in the parallel construct inherits the same priority of the parallel process. The first process in the priparallel process list will get the highest priority and the last process in the process list will get the lowest priority of the priparallel construct. The priparallel object itself is a process.

Currently, the maximum number of priorities per priparallel is 8, where 7 for user defined processes and one reserved for an idle task, skip task, garbage collector task (not implemented). The reserved priority is private to the priparallel. The restriction of the maximum of 8 priorities enables fast priority sorting with the efficiency of order $O(2)$. The process can be stored into the right priority queue in a maximum of two steps. PriParallel compositions may be nested within other priparallel processes to arbitrary length. This will be covered later.

The processes are executed by priority, but they don't have a priority by itself; in other words, the user cannot assign a priority to a process instead they may add the processes to a priparallel construct. The philosophy 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 and not by some number. At implementation level, the priparallel process will solve the ordering of the priorities.

The `PriParallel` class creates the priority based parallel composition construct.

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

The argument `processes` is an array of processes. The construct starts by invoking the `run()` method.

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

The following example shows a 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 successive priorities. Process `Process1` (at index 0) has the highest priority (=0). All processes in the process list with index 6 and higher share the lowest priority (=6). The `pripar` process terminates successfully when all three processes are successfully terminated.

Increasing the maximum number of priorities (larger than 7) is possible by nesting a second priparallel process in a priparallel process. The following example illustrates a priparallel construct of 49 ($=7^2$) priorities.

```
PriParallel pripar = new PriParallel(new Process[]
{
  new PriParallel(new Process[]                    // priority 0
  {
    Process1_1(channel interfaces),                // priority 0.1
    ..                                             // priority 0.2-6
    Process1_7(channel interfaces)                 // priority 0.7
  }),
  new PriParallel(new Process[]                    // priority 1
  {
    Process2_1(channel interfaces),                // priority 1.1
    ..                                             // priority 1.2-6
    Process2_7(channel interfaces)                 // priority 1.7
  }),
  new PriParallel(new Process[] { idem }),         // priority 2.1-2.7
  new PriParallel(new Process[] { idem }),         // priority 3.1-3.7
  new PriParallel(new Process[] { idem }),         // priority 4.1-4.7
  new PriParallel(new Process[] { idem }),         // priority 5.1-5.7
  new PriParallel(new Process[] { idem })          // priority 6.1-6.7
});

pripar.run();
```

New processes can be added at run-time, by

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

or by adding multiple processes at once

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

A new process can be inserted at run-time by

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

Process `process` will be inserted at `index` of the process list.

A process can be removed from the process list, by

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

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

5.4 The alternative composition construct - Alternative

The alternative composition construct consists of guards that in turn each guard a process. As soon as a guard becomes ready it is executed followed by the guarded process. This completes the execution of the alternative construct. The `Alternative` class defines the alternative composition construct. The alternative object itself is also a process.

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

```
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();
```

The `alt` process waits until at least one guard becomes ready, but terminates successfully when one of the three processes is selected and is successfully terminated. The guard with $\texttt{Process}_i$ will be selected when $\texttt{channel}_i$ is ready. Here, $\texttt{channel}_i$ is an input channel of $\texttt{Process}_i$. If more than one guard is ready than one guard will be 'randomly' selected; theoretically, by a non-deterministic choice. This alternative construct is *fair* selective, i.e. when more than one guard is ready then one ready guard will be selected according to a circular mechanism. The process of the selected ready guard will be executed.

New guards can be added at run-time, by

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

or by adding multiple guards at once

```
alt.add(new Guard[]
{
  new Guard(channel4, new Process4(channel4, ..),
  new Guard(channel5, new Process5(channel5, ..)
});
```

A guard can be removed from the guard list, by

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

5.4.1 Unconditional and conditional guards

The guard object guards a process by signaling the alternative construct that the first occurrence of a communication input event of the process is ready. The process is input guarded i.e. only input channels can be monitored by the guard. Output guarding of a process is not implemented because this would result in a significant penalty of the channel performance. On the other hand, input guards and output guards, at both ends of a channel, will never meet their ready state. Therefore, eliminating output guards avoids this hazardous situation, simplifies the channel implementation, and increases the performance. For most application, input guarding is sufficient [12].

A guard object can be declared as follows,

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

The guard becomes `true` when argument `channel` (input interface) is ready and has data available to be read by `Process`. The guard itself is not a process, but a passive object that guards a process. The alternative object will check all guards for readiness and waits until at least one channel becomes ready. The process belonging to a ready guard may be selected and executed. The above guard always participates in the alternative construct and is called *unconditional*. A guard may also be *conditional* i.e. the guard is enabled – participates in the

alternative construct – if some condition is true; otherwise the guard is disabled – omitted by the alternative construct – and the process will not be selected.

```
boolean condition = true;
```

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

If `condition` is `true` the guard will check `channel`, otherwise the guard is omitted and the process will not be selected. Once the process is selected, the process must read on `channel`.

A guard declared as

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

is equal to

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

The condition of the guard can be changed by the `setEnabled()` method.

```
guard.setEnabled(false);
```

Applying the conditional guards is useful for implementing state transitions.

There are also alternative constructs possible based on SKIP and TIMEOUT.

5.4.2 The alternative composition construct with SKIP

The alternative construct may continue (SKIP) when no channel is ready.

```
Alternative alt = new Alternative(guards, enable, skip_process);
```

The argument `guards` is the array of guard objects. If argument `enable` is `true` than the specified `skip_process` will be executed when no channel is ready. The same applies for the prialternative construct.

5.4.3 The alternative composition construct with TIMEOUT

The alternative construct may continue after some time (TIMEOUT) when no channel becomes ready.

```
Alternative alt = new Alternative(guards, msec, nsec, enable,
timeout_process);
```

The argument `guards` is the array of guard objects. The arguments `msec` and `nsec` specify the timeout in milliseconds plus nanoseconds. If argument `enable` is `true` than the specified `timeout_process` will be executed when no channel is ready after the timeout has expired. The same applies for the prialternative construct.

5.5 The priority based alternative composition construct - PriAlternative

The `PriAlternative` class creates the priority based alternative composition construct. The `PriAlternative` class extends the `Alternative` class and overrides the ‘fair’ choice mechanism with a ‘priority’ based choice mechanism. The prialternative object itself is also a process. The prialternative construct is similar to the alternative construct.

```
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, but terminates successfully when one of the three processes is selected and is successfully terminated. The guard with $\texttt{Process}_i$ will be selected when $\texttt{channel}_i$ is ready. Here, $\texttt{channel}_i$ is an input channel of $\texttt{Process}_i$. If more than one guard is ready than the guard with the lowest index will be selected. The process of the selected ready guard will be executed.

New guards can be added at run-time, by

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

or by adding multiple guards at once

```
prialt.add(new Guard[]
{
  new Guard(channel4, new Process4(channel4, ..),
  new Guard(channel5, new Process5(channel5, ..)
});
```

A new guard can be inserted at run-time, by

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

Guard `guard` will be inserted at `index` of the guard list. The priority of `guard` and other guard below will automatically be incremented or decremented.

A process can be removed from the process list, by

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

5.6 Nested composition constructs

The `Sequential`, `Parallel`, `PriParallel`, `Alternative` and `PriAlternative` composition processes can be nested on other composition processes. For instance two sequential constructs running 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 running 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();
```

In the same way, alternative constructs can be nested as illustrated in the following example:

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

process.run();
```

6 Conclusions

The *Communicating Threads for Java* (CTJ) package is an implementation of the CSP model resulting in the use of much simpler and more reliable thread constructs, based on processes, compositions and channels, than the Java thread model. The CTJ package provides a small set of design patterns that is sufficient for 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, starvation etc. during design and implementation.

Reasoning about the behavior of processes becomes abstract and cognitive 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 becomes more simplified than digging in threads.

The concept of processes, channels and link drivers specifies a logical separation of hardware dependent and hardware independent disciplines. This framework may pay off when developing Java software for real-time and embedded systems in that there is a clear mapping between software and hardware. Subsequently, the concept embraces a clear path from the conceptual design down to the details of implementation.

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.