CSP for Java programmers, Part 2

Concurrent programming with JCSP

Abhijit Belapurkar June 21, 2005

In this second installment of his three-part introduction to Communicating Sequential Processes (CSP) for Java programmers, Abhijit Belapurkar shows you how to use the Java-based JCSP library to write multithreaded Java applications that are guaranteed to be free of concurrency issues such as race hazards, deadlocks, livelocks, and resource starvation.

CSP is a paradigm for modeling complex interactions between concurrent objects. One of the chief advantages of using CSP is the ability to precisely specify and verify the behaviors of the objects involved at each stage of a program. The theory and practice of CSP has had a profound impact on the areas of concurrent design and programming. It is the basis of programming languages like occam and has been influential in the design of others, such as Ada. As briefly discussed in Part 1 of this article, CSP can also be invaluable to Java developers for its applicability to safe and elegant multithreaded programming on the Java platform.

In this second part of my three-part introduction to CSP programming on the Java platform, I focus on the theory and practice of CSP, particularly as it applies to multithreaded programming in the Java language. I start with an overview of the theory of CSP and then introduce you to the Java-based JCSP library implementation, which has CSP engineered into it.

CSP basics

The basic constructs in CSP are processes and various forms of communication between them. Everything in CSP is a process, even a network of (sub)processes. However, there are no direct interactions between processes -- all interactions happen only through CSP synchronization objects, such as communication channels and event barriers to which groups of processes subscribe.

A CSP *process* is different from a typical Java object in that both the data encapsulated in a process component *and* the algorithms for manipulating that data are private. That is, processes do not have externally invokable methods (except of course for the one method that must be invoked to kickstart the process) and algorithms can only be executed by the process in its own thread of control. If you contrast this with a method invocation in the Java language, you can see immediately how CSP removes the need for explicit locking.

Don't miss the rest of the series!

"CSP for Java programmers" is a three-part introduction to Communicating Sequential Processes (CSP), a paradigm for concurrent programming that honors its complexity without abandoning you to it. Read the other parts of the series:
Part 1: Pitfalls of multithreaded programming on the Java platform
Part 3: Advanced topics in JCSP

In the Java language, a method invoked on an object always runs in the thread of the caller. A particular thread of control works its way through multiple objects in the system. Objects, for the most part, don't have a life of their own -- they only briefly come alive as running threads invoke methods on them. Because of this, different threads of execution can end up trying to invoke the same method on the same object at the same time, as discussed in Part 1. Clearly, this can never happen in CSP.

Communication channels and process networks

The simplest mechanism for interprocess communication is reading and writing data across channels. The basic channel construct in CSP is *synchronous* and *point-to-point*; that is, containing no internal buffering and connecting one process to another process. Starting from this basic channel, it is possible to construct multiple reader/writer channels (that is, one-to-any, any-to-one, and any-to-any) as well.

Processes in CSP form the basic building blocks for complex systems -- one process can be wired up with one or more other processes (all set to execute in parallel) to form a network of processes. This network itself can be thought of as a process that can, recursively, be combined with other processes, themselves networks or otherwise, into a myriad of complex arrangements designed to best solve the problem at hand.

Considered individually, a process is just an independent serial program that interacts only with external I/O devices. This program doesn't need to concern itself with the existence or the nature of a process at the other end of the I/O channel.

The theory of CSP has been implemented in a number of Java-based frameworks, including the Communicating Sequential Processes for Java (JCSP) library.

Learn more about CSP

This article provides a general introduction to the complex topic of CSP. If you're interested in getting down and dirty with the mathematics underlying the theory, you may want to refer to the original paper by C.A.R. Hoare, as well as the book he wrote on the subject. For a more recent account of the theory of CSP (which has been updated over the years) see the book by Bill Roscoe. For further reference from a wide variety of sources, check out the CSP Archive at the Oxford University Computer Laboratory and the WoTUG homepage. See Related topics for links to all of these references and more.

The JCSP library

The JCSP library was developed by Professor Peter Welch and Paul Austin at the University of Kent at Canterbury, UK (see Related topics). For most of the remainder of the article, I will focus

on how CSP concepts are implemented in JCSP. Because the Java language doesn't offer native support for CSP constructs, the JCSP library internally uses the native concurrency constructs the language *does* support, such as `synchronized`, `wait`, and `notify`. To help you understand exactly how JCSP works, I explain the internal implementation of some of the JCSP library classes in terms of these Java constructs.

Note that the examples in the following sections are based on and/or derived from those documented in the javadocs for the JCSP library and/or the presentation slides available on the JCSP home page.

Processes in JCSP

In JCSP, a process is essentially a class that implements the `CSProcess` interface. Listing 1 shows the interface:

Listing 1. The CSProcess interface

```
package jcsp.lang;

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

Notice that the `CSProcess` interface looks exactly like the Java language's `Runnable` interface and serves a similar role. Although JCSP is currently implemented using the standard Java API, it need not be, and in the future it may well not be. For this reason, the `Runnable` interface is not used directly in JCSP.

Verifying JCSP programs

Professor Peter Welch and others have built a formal CSP model that makes it possible to analyze any multithreaded Java application in CSP terms and verify that it is free of the bugs that can cause race hazards, deadlocks, and resource starvation. Because the JCSP library uses the monitor mechanisms underlying the model (namely `synchronized()`, `wait()`, `notify()`, and `notifyAll()`) JCSP-based applications are verifiable using various software engineering tools, including some commercially supported ones. See Related topics to learn about FDR2, a model-checking tool for CSP-based programs.

JCSP defines two interfaces for reading from and writing to channels. The interface to read from a channel is called `ChannelInput` and consists of a single method called `read()`. A process that calls this method on an object implementing the `ChannelInput` interface will block until an object is actually written on the channel by a process at the other end of the channel. Once such an object is available on the channel, it is returned to the calling process. Similarly, the `ChannelOutput` interface consists of a single method called `write(Object o)`. A process that calls this method on an object implementing the `ChannelOutput` interface will block until the object is accepted by the channel. As previously mentioned, the simplest type of channel doesn't have any buffering, and so it won't accept the object until the process on its other (reading) end calls `read()`.

From here on, I use code examples to demonstrate how these and other JCSP constructs work. In Listing 2, you can see a very simple process that outputs all even integers between 1 and 100:

Listing 2. Process to generate even integers between 1 and 100

```
import jcsp.lang.*;

public class SendEvenIntsProcess implements CSProcess
{
    private ChannelOutput out;

    public SendEvenIntsProcess(ChannelOutput out)
    {
      this.out = out;
    }

    public void run()
    {
      for (int i = 2; i <= 100; i = i + 2)
      {
        out.write (new Integer (i));
      }
    }
}
```

There must be a reading process corresponding to each writing process. The absence of such a process would cause the `SendEvenIntsProcess` to block indefinitely, immediately following the first write operation on the `ChannelOutput` object `out`. Listing 3 shows a simple read process that corresponds to the write process shown in Listing 2:

Listing 3. The corresponding consumer process

```
import jcsp.lang.*;

public class ReadEvenIntsProcess implements CSProcess
{
    private ChannelInput in;
    public ReadEvenIntsProcess(ChannelInput in)
    {
      this.in = in;
    }

    public void run()
    {
      while (true)
      {
        Integer d = (Integer)in.read();
        System.out.println("Read: " + d.intValue());
      }
    }
}
```

Channels in JCSP

At this point, all I have are two independent processes. My next step is to wire them up using a common channel as a shared synchronization mechanism and then kick each one off. A *channel* interface is a subinterface of JCSP's `ChannelInput` and `ChannelOutput` interfaces and is a common interface for reading and writing objects. There are many possible implementations of this interface, as described below:

- The class `One2OneChannel`, as the name indicates, implements a "single-writer-single-reader" type channel.

- The class `One2AnyChannel` implements a "single-writer-multiple-readers" object channel. (Note: This is not a broadcast mechanism in that the multiple readers actually compete with each other for reading from the channel; only one reader can use the channel along with the writer at any given time.)
- The class `Any2OneChannel` implements a "multiple-writers-single-reader" object channel. As in the above case, writing processes compete with each other to use the channel. Only the reader and one of the multiple writers can actually use the channel at any given time.
- The class `Any2AnyChannel` implements a "multiple-writers-multiple-readers" object channel. Reading processes compete with each other to use the channel, as do writing processes. Only one reader and one writer can actually use the channel at any one time.

In the example in Listing 3, I only have a single writer and a single reader process, so the `One2OneChannel` class will suffice. The sample code for the driver program is shown in Listing 4:

Listing 4. The driver program

```
import jcsp.lang.*;

public class DriverProgram
{
    public static void main(String[] args)
    {
      One2OneChannel chan = new One2OneChannel();
      new Parallel
      (
        new CSProcess[]
     {
       new SendEvenIntsProcess (chan),
       new ReadEvenIntsProcess (chan)
     }
      ).run ();
    }
}
```

As the code indicates, I first instantiate a new `One2OneChannel` object and then pass it to the constructors of the `SendEvenIntsProcess` and `ReadEventIntsProcess` processes. Doing it this way works because the `One2OneChannel` implements both interfaces -- `ChannelInput` and `ChannelOutput`.

Channel internals

Because channels are an important concept in JCSP, let's make sure you understand how they work before going further. As I mentioned before, channels are by default non-buffering, but they can be made buffering as well. This is made possible by the channels not dealing with buffering characteristics themselves and delegating that responsibility to another class, which must implement an interface called `ChannelDataStore`. JCSP provides multiple built-in implementations for this interface, including the following:

- `ZeroBuffer`, which corresponds to the default non-buffering characteristic.
- `Buffer`, which provides a blocking FIFO-buffered semantics for the channel with which it is associated.

- `InfiniteBuffer`, which also provides a FIFO semantics, except that only readers can be blocked if the buffer is empty. Writers are never blocked because the buffer capacity can be expanded indefinitely, or at least until the limit imposed by the underlying memory system is reached.

Channels at work

Consider an example of channels at work. When I created the `One2OneChannel` instance shown in Listing 4, I set its internal `ChannelDatasource` to a new instance of `ZeroBuffer`. `ZeroBuffer` can store only one object (or integer). It has an internal state variable that starts with the value `EMPTY` and becomes `FULL` as soon as an object is put into it.

What happens when the `SendEvenIntsProcess` process does a `write` on its *out* channel? Well, the `write()` method of the `One2OneChannel` class is a `synchronized()` method. Therefore, the thread on which the sender process is running (as you'll see shortly the sender and reader processes run on separate threads) acquires the monitor lock associated with this channel instance and proceeds with the method. Within the method, the first order of business is to write the object (or, in this case, the integer) to the `ZeroBuffer` instance held internally, by calling the `put` method on it. This changes the state of the buffer to `FULL`. At this point, the calling thread invokes `wait`, causing the thread to enter the *wait set* of the monitor, which is followed by the monitor lock getting released and the thread getting blocked.

At a later point in time, the reader thread invokes the `read` operation on the channel (this is also a synchronized method, so the reader thread must acquire the monitor lock before proceeding). Because the state of the internal buffer is `FULL`, the available data is returned and a `notify()` issued. This `notify()` wakes up the sender thread, which then exits the monitor wait set and reclaims the monitor lock.

In the converse scenario, if the reader thread had invoked the `read` method on a channel whose internal buffer was in the `EMPTY` state, it would have had to `wait`, in which case the sender thread would have notified it after writing the data object into the internal buffer.

The Parallel construct

You may have noted in Listing 4 that the driver program introduced a new class called `Parallel`. The `Parallel` class is provided by JCSP as a predefined `CSProcess` that takes an array of individual `CSProcess` instances and runs them "in parallel" (all processes except the last one are run in distinct threads; the last process is run by the `Parallel` object in its own thread of control). A `run` of a `Parallel` process terminates when, and only when, all its component processes terminate. A `Parallel` process is therefore a mechanism for composing multiple individual processes together, using channels (in the driver program example) as "wires" to connect them.

Another way of looking at the `Parallel` construct is to say that it makes it possible to compose a higher-level process from smaller and simpler components. In fact, `Parallel` lets you create an entire *connected network of processes* of arbitrary complexity over several iterations, by wiring together components created in each previous iteration in each new one. The resulting network of processes can be exposed and used as yet another `CSProcess` object.

A Parallel example

The JCSP library provides a collection of plug-and-play components that are intended to serve an educational purpose only, which suits my purposes just fine: delving into the internal implementation of a few of them should nicely demonstrate how networked concurrent processes can be composed in JCSP. I use the following example processes to illustrate the inner workings of the `Parallel` construct in JCSP:

- `PlusInt` takes in integers on its two input streams, adds them together, and outputs the result to its output stream.
- `Delta2Int` broadcasts, in parallel, every integer arriving on its input streams to its two output channels.
- `PrefixInt` prefixes a (user-configured) integer to its integer input stream. (That is, the first output from this process, before any integer is available on its input channel, is the configured integer itself. The following outputs are the integers taken as-is from the input stream.)
- `IntegrateInt` is a process composed of the previous three using the `Parallel` construct. Its function is to output running totals of the integers coming down its input channel.

The `run` method of the `IntegrateInt` class is shown in Listing 5:

Listing 5. The IntegrateInt process

```
import jcsp.lang.*;

public class IntegrateInt implements CSProcess
{
  private final ChannelInputInt in;
  private final ChannelOutputInt out;

  public IntegrateInt (ChannelInputInt in, ChannelOutputInt out)
  {
    this.in = in;
    this.out = out;
  }

  public void run()
  {
      One2OneChannelInt a = new One2OneChannelInt ();
      One2OneChannelInt b = new One2OneChannelInt ();
      One2OneChannelInt c = new One2OneChannelInt ();

      new Parallel
      (
        new CSProcess[]
        {
          new PlusInt (in, c, a),
          new Delta2Int (a, out, b),
          new PrefixInt (0, b, c)
        }
      ).run ();
  }
}
```

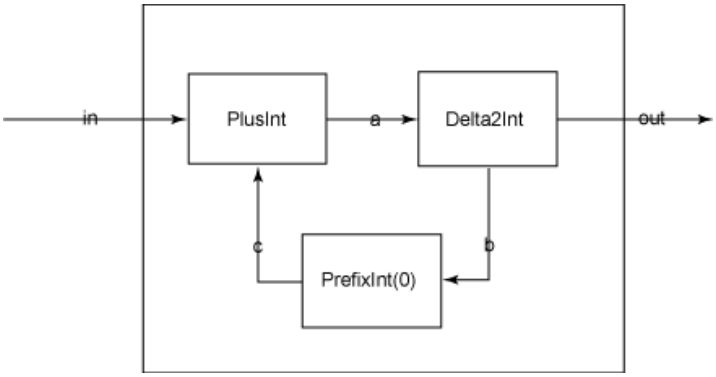
Note the use of different kinds of channels in this example as compared to those shown in Listing 4. The `IntegrateInt` class uses `ChannelInputInt` and `ChannelOutputInt` channels that, as their

names indicate, can be used to transfer integers of type `int`. In contrast, the driver program in Listing 4 used `ChannelInput` and `ChannelOutput`, which are *object* channels that can be used to send any arbitrary object down the channel from the sender to the receiver. For this reason I had to wrap `int` values as `Integer` objects before transferring them in Listing 4.

What else do you notice looking at the code in Listing 5? Essentially, the first output from the `PrefixInt` process is 0, which is added by the `PlusInt` process to the first integer to arrive on the input channel. This result is written to the channel *a*, which forms the input channel for the `Delta2Int` process. The `Delta2Int` process writes the integer result onto *out* (the overall output channel for the process) and sends it to the `PrefixInt` process. The `PrefixInt` process then sends the integer as-is input to the `PlusInt` process to be added to the second integer in the stream, and so on.

A diagrammatic representation of the composition of the `IntegrateInt` process is shown in Figure 1:

Figure 1. The IntegrateInt process



Networks within networks

The `IntegrateInt` process, thus composed from three smaller processes, can itself be used as a composing process. The JCSP library provides a process called `SquaresInt` which, as the name indicates, generates a stream of integers that are squares of natural numbers (1, 2, 3, 4, and so on). The code for this process is shown in Listing 6:

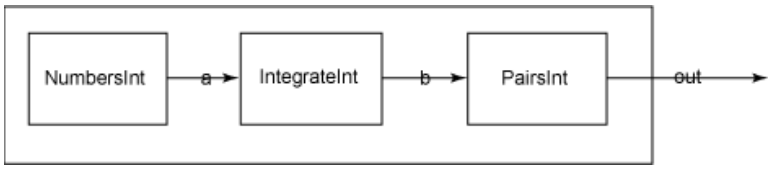
Listing 6. The SquaresInt process

```
public class SquaresInt implements CSProcess
{
  private final ChannelOutputInt out;

  public SquaresInt (ChannelOutputInt out)
  {
    this.out = out;
  }

  public void run()
  {
      One2OneChannelInt a = new One2OneChannelInt ();
      One2OneChannelInt b = new One2OneChannelInt ();
```

```
      new Parallel
      (
        new CSProcess[]
        {
          new NumbersInt (a),
          new IntegrateInt (a, b),
          new PairsInt (b, out)
        }
      ).run ();
  }
}
```

I'm sure you've noted two new processes in Listing 6. `NumbersInt` is a built-in process that simply outputs natural numbers starting from 0 on its output channel. `PairsInt` is a process that adds successive pairs of input values and outputs the result. These two new processes along with `IntegrateInt` comprise the `SquaresInt` process, as illustrated by the diagram in Figure 2:

Figure 2. The SquaresInt process



How SquaresInt works

Let's consider the internal workings of the `SquaresInt` process before moving on. Below you can see how the traffic flowing on the individual channels within `SquaresInt` looks:

```
Channel "a": [0, 1, 2, 3, 4, 5, 6, 7, 8, ...ad infinitum]
Channel "b": [0, 1, 3, 6, 10, 15, 21, 28, 36, ...ad infinitum]
Channel "out": [1, 4, 9, 16, 25, 36, 49, 64, 81 ...ad infinitum]
```

Do you see the pattern of how the integers being written onto channel *a* lead to those being written onto channel *b*, and thereby onto channel *out*? In the very first "tick," the `NumbersInt` process writes the integer 0 onto channel *a*. The `IntegrateInt` process also writes the integer 0 (which is, after all, the current value of the running total) onto channel *b*. The `PairsInt` process doesn't produce anything on this tick because it needs two inputs to work with. Upon the second tick, the `NumbersInt` process writes the integer 1 onto its ouput channel. This causes the `IntegrateInt` process to change the running total to `0+1=1` and thereby write the integer 1 onto channel *b*.

At this point, `PairsInt` has two integer inputs to work with -- the integer 0 from the previous tick and the integer 1 from the present one. It adds them together and writes the output `0+1=1` onto channel *out*. Note that 1 is the square of 1, so we're probably on the right track here. Taking the example forward to the next (third) tick, the `NumbersInt` process writes the integer 2 onto channel *a*. This makes the `IntegrateInt` process update the running total to `1` (the previous total) `+ 2` (the new value) `= 3` and write this integer out onto channel *b*.

What are the last two integers seen by the `PairsInt` process? They are `1` (during the previous tick) and `3` (during the present tick). Therefore, the process adds these two integers and writes `1+3=4` onto channel *out*. You will notice that 4 is the square of 2, indicating that `SquaresInt` is

working as it should. In fact, you could continue to run the program for any arbitrary number of ticks henceforth, and you would be able to verify that the integer written onto channel *out* was always a square of the next-in-line integer. I do precisely that in the next section.

A mathematical digression

Just in case you're wondering, I can explain the mathematical basis of how the squares are generated. Suppose you had peeked inside the box at the time when the `NumbersInt` process had already output integers up to a certain *n-1*. The running sum that was last produced by the `IntegrateInt` process (and fed into `PairsInt` process through the shared channel *b*) would be `[1+2+3+...+(n-1)] = (n-1)(n-2)/2`.

During the next tick, `NumbersInt` would output *n*, causing the running sum in the `IntegrateInt` process to increase to `(1+2+3+...+n) = n(n-1)/2`. This sum would then be fed to the `PairsInt` process through the shared channel *b*. `PairsInt` would add these two numbers together to generate `[(n-1)(n-2)/2 + n(n-1)/2] = [(n-2) + n](n-1)/2 = (2n-2)(n-1)/2 = (n-1)exp2`.

Next, the `NumbersInt` process would produce *(n+1)*. Corresponding to this, the `IntegrateInt` process would feed `n(n+1)/2` to the `PairsInt` process. `PairsInt` would then generate `[n(n-1)/2 + n(n+1)/2] = nexp2`. Generalizing this for all *n* would yield all the squares, as desired.

Determinism in JCSP

The above examples demonstrate the compositional semantics of CSP -- namely how the `Parallel` construct can be used to compose layered networks from fine-grained stateless components. The selling point for all such layered networks of communicating parallel processes is that they are completely deterministic. What does *deterministic* mean in this context? It means that the output(s) from such a layered network depend only on the inputs provided to it, regardless of the characteristics of the run-time environment (JVM) the network runs in. That is, the process network is independent of the scheduling policies of the JVM and also of its distribution onto multiple processors. (I'm assuming a single node here; however, nothing inherently prevents this same argument from being carried over to a process network that is physically distributed across multiple nodes with processes communicating on-the-wire.)

Determinism is a powerful tool to have in your kit because it enables you to cleanly reason about the behavior of your program, without having to worry about what effect the run-time environment may have on it. At the same time, determinism isn't the only possible or necessary approach to concurrent programming. As the next (and final) working example will show, non-determinism is an equally powerful concept-at-work in the JCSP library.

Non-determinism in JCSP

Non-determinism is a factor in many real-life applications where the visible outcome is a function of the order in which events happen. Put another way, non-determinism comes into play in concurrent applications where the result is dependent on scheduling by design, and *not* by accident. JCSP handles such issues explicitly, as you'll see.

Suppose, for example, that a process has a number of alternatives to choose from in terms of *what to do next*. Each alternative has a *guard* associated with it, which must be "ready" in order for the alternative to be considered for possible selection. The process selects one alternative from among the available (that is, ready) options; the selection itself may be based on different strategies, namely arbitrary selection, highest priority selection, or fair selection.

Event selection strategies

In the specific context of JCSP, an abstract class called `Guard` is provided, which must be subclassed by event objects that are competing for selection by a process. The process itself uses another preprovided class called `Alternative`, to whose constructor these guard objects must be passed in as an object array. The `Alternative` class provides methods for the three types of event-selection strategy.

The method `select()` of the `Alternative` class corresponds to an *arbitrary selection* strategy. The `select()` method call blocks until one or more guards are ready (remember that all contending guards are known to the `Alternative` class). One among the ready guards is arbitrarily selected and its index (in the array of guards passed in) is returned.

The method `priSelect()` corresponds to the *highest priority* strategy. That is, if more than one guard is ready the one with the lowest index is returned; the assumption being that guards in the array passed to the `Alternative` constructor have already been sorted in decreasing order of priority.

Lastly, the method `fairSelect` is *fair* in choosing between the more-than-one ready guards: In successive invocations of this method, no single ready guard will be selected twice while another ready and available guard goes unselected. Therefore, if the total number of guards is *n*, no ready guard will go unselected for more than *n* successive selection operations, in the worst case.

The arbitrary selection strategy works best for processes that do not care, if multiple guards are ready, how the selection is made; it does not work so well for processes that want to guarantee no starvation or worst-case service times, such as in real-time systems. In the former case, using the method `fairSelect` is recommended, while in the latter, using the method `priSelect()` is best.

Types of guards

Broadly, JCSP provides three types of guards:

- **Channel guards** always correspond to a channel that the process is waiting to read data from. That is, a channel guard is ready if and only if the process at the other end of the channel has output to it and this data has not yet been input by this process.
- **Timer guards** always correspond to setting (absolute) timeouts. That is, a timer guard is ready if its timeout has expired.
- **Skip guards** are always ready.

A *channel guard* in JCSP can be of the following types:
`AltingChannelInput`/`AltingChannelInputInt`, which are ready whenever object/integer data

(respectively) is pending in the corresponding channel; or `AltingChannelAccept`, which is ready if an unaccepted "CALL" is pending on the channel (more on this to follow). These are abstract classes and have concrete implementations in the form of `One2One` and `Any2One` type channels. A *timer* guard in JCSP is of the type `CSTimer` while a *skip guard* is provided as the `Skip` class.

Guards at work

I conclude this introduction to JCSP with a simple example that demonstrates the use of JCSP guards to facilitate non-determinism in concurrent applications. Suppose you have to develop a multiplication (or *scaling*) device that reads integers arriving at a fixed rate on its input channel, multiplies them by a certain factor, and writes them out to its output channel. The device can start with an initial factor, but this factor is doubled automatically every five seconds.

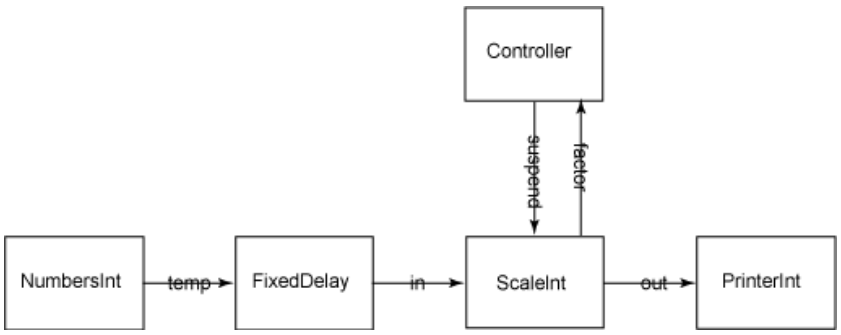
The twist in the tale is as follows: A second controller process exists in the system that can send a `suspend operation` signal to your device over a dedicated channel. This makes the device suspend itself and send the current value of the multiplying factor to the controller over a second channel.

When suspended, the device should simply allow all incoming integers to pass through unchanged on its output channel. The controller process -- perhaps after performing some computations using as input the multiplying factor sent to it by the device -- sends a new factor down to the device over a dedicated channel. (Note that your device is obliged to accept this factor whenever it is in the *suspended* state.)

The injection of the updated factor into your device also serves as its wakeup signal. The device now resumes its scaling-up operation, multiplying the input integers by the newly updated factor. The timer is also reset at this point, so the new multiplying factor is set to be doubled after five seconds, and so on.

The diagram in Figure 3 depicts the scaling device:

Figure 3. The scaling device



The ScaleInt process

The source code for the scaling device is shown in Listing 7. The non-determinism in this example is because of the fact that the *output* value is based on the values on the *in* and *inject* streams (and also on the sequence in which these values arrive).

Listing 7. The ScaleInt process

```
import jcsp.lang.*;
import jcsp.plugNplay.ints.*;

public class ScaleInt implements CSProcess
{
  private int s;
  private final ChannelOutputInt out, factor;
  private final AltingChannelInputInt in, suspend, inject;

  public ScaleInt (int s, AltingChannelInputInt suspend, AltingChannelInputInt in,
    ChannelOutputInt factor, AltingChannelInputInt inject, ChannelOutputInt out)
  {
    this.s = s;
 this.in = in;
 this.out = out;
 this.suspend = suspend;
 this.factor = factor;
 this.inject = inject;
  }

  public void run()
  {
 final long second = 1000;               // Java timings are in millisecs
 final long doubleInterval = 5*second;
 final CSTimer timer = new CSTimer ();

 final Alternative normalAlt = new Alternative (new Guard[] {suspend, timer, in});

 final int NORMAL_SUSPEND=0, NORMAL_TIMER=1, NORMAL_IN = 2;

 final Alternative suspendedAlt = new Alternative (new Guard[] {inject, in});

 final int SUSPENDED_INJECT=0, SUSPENDED_IN = 1;

 long timeout = timer.read () + doubleInterval;
 timer.setAlarm (timeout);

 while (true)
 {
   switch (normalAlt.priSelect ())
   {
  case NORMAL_SUSPEND:
    suspend.read ();              // don't care what's sent
    factor.write (s);             // reply with the crucial information
    boolean suspended = true;
    while (suspended)
    {
      switch (suspendedAlt.priSelect ())
   {
     case SUSPENDED_INJECT:    // this is the resume signal as well
       s = inject.read ();     // get the new scaling factor
    suspended = false;      // and resume normal operations
    timeout = timer.read () + doubleInterval;
    timer.setAlarm (timeout);
    break;
     case SUSPENDED_IN:
       out.write (in.read ());
    break;
   }
    }
    break;
  case NORMAL_TIMER:
    timeout = timer.read () + doubleInterval;
    timer.setAlarm (timeout);
    s = s*2;
```

```
    break;
  case NORMAL_IN:
    out.write (s * in.read ());
    break;
   }
    }
  }
}

import jcsp.lang.*;
import jcsp.plugNplay.ints.*;

public class Controller implements CSProcess
{
  private long interval;
  private final ChannelOutputInt suspend, inject;
  private final ChannelInputInt factor;

  public Controller (long interval, ChannelOutputInt suspend, ChannelOutputInt inject,
    ChannelInputInt factor)
  {
    this.interval = interval;
    this.suspend = suspend;
    this.inject = inject;
    this.factor = factor;
  }

  public void run ()
  {
 int currFactor = 0;
 final CSTimer tim = new CSTimer ();
 long timeout = tim.read ();
 while (true)
 {
   timeout += interval;
   tim.after (timeout);        // blocks until timeout reached
   suspend.write (0);          // suspend signal (value irrelevant)
   currFactor = factor.read ();
   currFactor ++;              // compute new factor
   inject.write (currFactor);  // inject new factor
 }
  }
}

import jcsp.lang.*;
import jcsp.plugNplay.ints.*;

public class DriverProgram
{
  public static void main(String args[])
  {
 try
 {
   final One2OneChannelInt temp = new One2OneChannelInt ();
   final One2OneChannelInt in = new One2OneChannelInt ();
   final One2OneChannelInt suspend = new One2OneChannelInt ();
   final One2OneChannelInt factor = new One2OneChannelInt ();
   final One2OneChannelInt inject = new One2OneChannelInt ();
   final One2OneChannelInt out = new One2OneChannelInt ();

   new Parallel
   (
  new CSProcess[]
  {
    new NumbersInt (temp),
    new FixedDelayInt (1000, temp, in),
    new ScaleInt (2, suspend, in, factor, inject, out),
```

```
    new Controller (6000, suspend, inject, factor),
    new PrinterInt (out, "--> ", "\n")
  }
   ).run ();
 }
 catch (Exception e)
 {
  e.printStackTrace();
 }
  }
}
```

The above class, `ScaleInt`, corresponds to the scaling device. As mentioned before, this class must implement the `CSProcess` interface. Because the above code demonstrates many concepts, I discuss its various aspects one by one.

Two Alternatives

The first method of interest in the `ScaleInt` class is `run()`. The first thing I've done inside the `run()` method is to create two instances of the `Alternative` class, each with a different array of `Guards` objects.

The first `Alternative` instance, denoted by the variable `normalAlt`, is intended for use when the device is operating normally. The list of guards associated with it is as follows:

- **suspend** is an instance of `One2OneChannelInt`. As mentioned before, a `One2OneChannelInt` implements a single reader/writer integer channel that is zero-buffered and fully synchronized. This is the channel on which the controller process sends the suspend signal to the device.
- **timer** is an instance of `CSTimer`, which is set to trigger every five seconds, at which time the device will double the current value of the multiplying factor.
- **in** is an instance of `One2OneChannelInt` on which the device receives input integers.

The second `Alternative` instance, denoted by `suspendedAlt`, is intended for use when the device has been previously suspended by the `Controller`. The list of guards associated with it is as follows:

- **inject** is an instance of `One2OneChannelInt` that will be used by the controller process to send a new multiplying factor (which also serves as a wake-up signal) to the device.
- **in** is the same instance of `One2OneChannelInt` you saw previously; the device receives input integers on this channel.

The two `Alternative` instances are used in different circumstances to wait for guards to get ready, with the order of listing being the implicit priority orders. For example, if the `suspend` and `timer` guards of `normalAlt` happen to be ready at the same time, the event corresponding to the `suspend` guard will be processed first.

On your guard(s)!

The next point of interest is what happens when each of the guards is ready. I deal with `normalSelect` first, assuming that the device is operating normally (that is, hasn't been suspended yet):

- If the controller has sent a *suspend* signal to the device, this event gets processed with the highest priority. In response, the device sends the current value of the multiplication factor to the controller over the channel called `factor`. It then sets an internal flag called `suspended` to `true` and enters a loop, waiting to be sent a signal to resume. Inside this loop, the device invokes the `priSelect()` method on the second `Alternative` instance (`suspendedAlt`). This `Alternative` instance consists of two guards: the first one denotes the event wherein the controller sends a new multiplication factor to the device and the second denotes the arrival of an integer on the input channel of the device. In the former case, the device updates the factor (held in variable `s`) with the new value read from the `inject` channel, sets the `suspended` flag back to `false` (thereby ensuring that the inner loop will be exited during the next iteration), resets the alarm using the current timer value as the base, and breaks out. In the latter case, the device simply reads the integer from its input channel and writes it out on the output channel (this is per the requirement that the multiplication factor not be used while the device is suspended).
- The event that gets processed with the next highest priority is the one where the alarm goes off. This causes the device to double the current multiplication factor, reset the alarm using the current timer value as the base, and go back to waiting for the next event.
- The third possible event is wherein an integer is received on the device's input channel. In response to this, the device reads the integer, multiplies it with the current factor `s`, and writes the result on its output channel.

The Controller class

The next class to consider is the `Controller` class. Remember, the job of the controller class is to inject new multiplying factor values into the device process on a periodic basis (based on a complex computation, presumably). In this example, the periodic basis is just a timer that goes off at a regular, configurable, interval. Every time that happens, the controller writes an *0* on the *suspend* channel (that is, it suspends the device) and reads the current multiplying factor on its input channel called `factor`.

At this point, the controller merely increments this value by one and injects it back into the device over the one-to-one channel (called `inject`) dedicated to this purpose. This signals the device to resume, at which point the timer is reset to go off after the appropriate interval.

The DriverProgram class

The only class that remains is the driver class `DriverProgram`. This class creates the appropriate channels and an array of `CSProcess` instances. It uses the JCSP-provided class `NumbersInt` to generate a sequence of natural numbers that are fed through the *temp* channel to another built-in class, called `FixedDelayInt`. As its name indicates, `FixedDelayInt` passes data coming in through its input channel onto its output channel after a fixed delay -- one second in the example code.

This stream of natural numbers one second apart is then fed into the *in* channel of the `ScaleInt` process. The output from the *out* channel of the `ScaleInt` process is fed to the JCSP-provided `PrinterInt` process, which in turn prints the integer values onto `System.out`.

Conclusion to Part 2

In this second part of my three-part introduction to CSP for Java programmers, I've both explained and demonstrated the theory of CSP in concurrent programming. Following an overview of CSP constructs, I introduced you to the most popular of the Java-based CSP libraries, JCSP. Since the Java language doesn't offer native support for CSP constructs, the JCSP library internally uses the native concurrency constructs that Java *does* support, such as `synchronized()`, `wait()`, and `notify()`. To help you understand exactly how JCSP works, I explained the internal implementation of some of the JCSP library classes in terms of these Java constructs, and then demonstrated their use in several working examples.

The discussion here serves as an excellent foundation for the final article in this series, where I explain the parallels between CSP and AOP, briefly compare the CSP approach to concurrency to that of the new `java.util.concurrent` package, and introduce you to a number of techniques for advanced synchronization with JCSP.

Acknowledgments

I would like to gratefully acknowledge the kind encouragement I received from Professor Peter Welch during the writing of this article series. His busy schedule notwithstanding, he took time to do a very thorough review of a draft version and gave many valuable inputs towards enhancing the quality and accuracy of the series. All remaining errors are mine alone! The examples I have worked with in my articles are based on and/or derived from those documented in the Javadocs for the JCSP library and/or the Powerpoint presentation slides available on the JCSP Web site. Both of these sources offer a wealth of information to be explored.

Related topics

- Brian Goetz's three-part "Threading lightly" is a smart and methodical approach to resolving synchronization issues on the Java platform (developerWorks, July 2001).
- C.A.R. Hoare's "Communicating Sequential Processes" introduced the parallel composition of communicating sequential processes as a fundamental program structuring method (*Communications of the ACM Archive*, 1978).
- C.A.R. Hoare's book on CSP is freely available in PDF format.
- Bill Roscoe's Theory and Practice of Concurrency (Prentice Hall, 1997) is an up-to-date book on the subjects of concurrency and CSP.
- The JCSP homepage is hosted by the University of Kent at Canterbury, UK.
- FDR2 (Failures-Divergence Refinement) is one of several commercially available model-checking tools for CSP-based programs.
- CSP implementations are available for languages other than Java: C++CSP is an implementation for C++.
- Occam-pi is a language platform intended to extend the CSP ideas of the occam language with the mobility features of the pi-calculus. Learn more about this cutting-edge research from the occam-pi homepage.
- While you're at it, you may also want to investigate the various extensions to the occam compiler.
- Also see the Java technology zone tutorials page for a complete listing of free Java-focused tutorials from developerWorks.