Process Oriented Design for Java: Concurrency for All

P.H.Welch
Computing Laboratory
University of Kent at Canterbury
CT2 7NF – England

Abstract: *Concurrency is thought to be an advanced topic – much harder than serial computing which, therefore, needs to be mastered first. This paper contends that this tradition is wrong, which has radical implications for the way we educate people in Computer Science – and on how we apply what we have learnt. A process-oriented design pattern for concurrency is presented with a specific binding for Java. It is based on the algebra of Communicating Sequential Processes (CSP) [1, 2, 3] as captured by the JCSP [4, 5, 6, 7, 8, 9] library of Java classes. No mathematical sophistication is needed to master it. The user gets the benefit of the sophistication underlying CSP simply by using it. Those benefits include the simplification wins we always thought concurrency should generate. Although the Java binding is new, fifteen years of working with students at Kent have shown that the ideas within process-oriented design can be quickly absorbed and applied. Getting the ideas across as soon as possible pays dividends – the later it's left, the more difficult it becomes to wean people off serial ways of thought that fit the world so badly. Concurrency for all (and for everyday use) in the design and implementation of most kinds of computer system is both achievable and necessary.*

Keywords: *processes, channels, design patterns, Java™, CSP, JCSP.*

1 Introduction

To be of use in the real world, computer systems need to model, at an appropriate level of abstraction, those parts of it for which they aim to be of service. Interesting things happen in the world as the result of the actions and interactions of vast numbers of independent agents (processes) operating at many levels of scale – from sub-atomic, through human to astronomic. If our computer modeling can reflect the natural concurrency in the system, it ought to be much simpler.

But concurrency, as traditionally presented and used, is very hard. The *monitor-threads* model provided by Java, whilst easy to understand in its primitives, proves very difficult to apply with confidence in any system above a modest level of complexity [5, 10]. Numerous warnings in Java textbooks (and on some of Sun's web pages) emphasize the difficulties of multi-threading (race hazards, deadlock, livelock and process starvation) and recommend getting involved only as a last resort. Teaching such models to first year undergraduates would not be a good idea.

However, concurrency is too powerful and, indeed, too simple an idea to be set aside. With a better handle, it can simplify both the design and the implementation of most complex systems, as well as boost performance. If this were not the case, nature would not have evolved the highly concurrent mechanisms we see all round us.

2 Objects Considered Harmful

An *object* encapsulates both *data* and the *methods* for inspecting and manipulating that data. An individual object is but one instance of a general *class*, of which there may be many other distinct instances. Classes are related to each other through an *inheritance* relation, whereby a sub-class extends the behavior of the super-class by adding data fields and methods and by changing the algorithms of existing methods. This is supposed to reflect a natural world order and is the philosophy that has revolutionized our approach to system design and implementation over the past decade.

Yet weaknesses are apparent in this philosophy, particularly in the context of concurrent systems.

Firstly, most objects are dead – they have no life of their own. All object methods have to be invoked directly (or indirectly) by an external thread of control – they have to be *caller-oriented* (a somewhat curious property of so-called *object-oriented* systems).

In serial object-oriented design, a single thread of control must snake around all objects in the system, bringing them to life *transiently* as their methods are executed. In a concurrent system, an object's methods (and, hence, state) are at the mercy of *any* thread that can see it. Nothing can be done to prevent method invocation, even when the invoked object is not in a fit state to receive it. The object is not in control of its life. Threads cut across object boundaries in spaghetti-like trails, *paying no regard to the underlying structure*.

Finally, data encapsulation breaks down all too easily. For one thing, the supposedly *private* attributes of an object may themselves be objects. Since all objects live in a universally accessible heap, those attributes may be shared between any number of other objects – sometimes by design but often by accident. Either way, individual control of an attribute is lost and, hence, local and simple reasoning about its properties.

Even when the attribute is a primitive data-type, we are not safe. *[I am indebted to Tom Locke for the following example.]* Consider a Java class, `X`, with a private integer field, `count`, and public methods that change it. Suppose the following lines of code occur in one of its methods:

```
count = 42;
thing.f ();
```

where `thing` is declared as an interface type that includes the `f()` method. What is the value of `count` after these two lines?

The answer is that we do not know! Suppose `me` is the object instance of `X` whose above two lines of code are being executed. Whoever constructed `me` may have given my reference to the actual object represented by `thing`. In which case, my invocation of its `f()` method may call `me` back and change the value of `count`.

This lack of ability to reason locally about key variables is strangely familiar. In the bad old days, free use of globals led us into exactly the same mess. *Structured programming* led us out that mire. Is *object-orientation* taking us back in?

Note that the above is not a problem introduced by concurrency. However, Java's *monitor* concepts do nothing to eliminate it. Suppose the `count` modifying methods were `synchronized`, along with the method containing the two code lines above. That certainly prevents *other* threads from interfering with the `count` variable during execution of the sequence – but allows the thread executing the sequence to re-enter the monitor when it calls `me` back and still make the change!

With inheritance, the problem deepens. Now, even if `thing` is an instance of a concrete class for which we have full documentation and source code and we can see that the `f()` method does not touch anything that could be `me`, we can still draw no conclusion about the value of `count`! The `thing` may actually be an instance of some *sub-class*, whose overridden `f()` method may do anything it likes.

The concept of *process orientation* described in the rest of this paper addresses these weaknesses in *OO*, at the same time providing an elementary and powerful model of concurrency.

3 Process Oriented Design

Concurrent behavior from the objects in a system ought to be our *normal expectation* – not something difficult that we add in as an *advanced feature* to improve user response times or other performance indicators. Concurrency should provide:

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

Java's in-built monitor concepts score badly on the above [5, 10]. We outline instead a model based on *Communicating Sequential Processes* (CSP) [1, 2, 3]. CSP is a mathematical theory for specifying and verifying complex patterns of behavior arising from interactions between concurrent objects. CSP has a formal and compositional semantics that lines up with our informal intuition about the way things work. No formal CSP algebra, therefore, need be presented to teach and use this model. Most ideas can be introduced through images of physical objects.

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

We can leave aside the formal mathematics, secure in the knowledge that it is mature and well-founded (and that some powerful model checking tools[11] – based on that mathematics – are available for when we need them).

3.1 Processes

A CSP *process* is a component that encapsulates data structures and algorithms for manipulating that data. Both its data and algorithms are private. The outside world can neither see that data nor execute those algorithms. It is not an *object*.

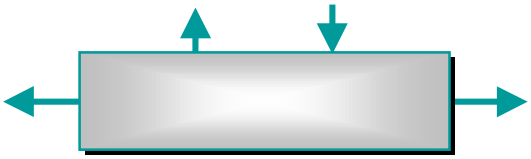

Figure 1: a process with a channel interface

Each process is alive, executing its own algorithms on its own data. Processes interact solely via CSP *synchronizing primitives* (such as *channels*) – not by calling each other's methods. Objects implementing those primitives form the *CSP interface* to a process (e.g. the channels in Figure 1).

3.2 Synchronizing Channels

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

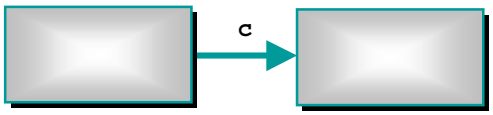

Figure 2: a simple network

In Figure 2, `A` and `B` are processes and `c` is a channel connecting them. Wires have no capacity to hold information, being only media for transmission. To avoid undetected loss of data, channel communication is synchronized. This means that if `A` transmits before `B` is ready to receive, then `A` will block. Similarly, if `B` tries to receive before `A` transmits, `B` will block. When both are ready, data is transferred – directly from the state space of `A` into the state space of `B`.

3.3 Networks

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

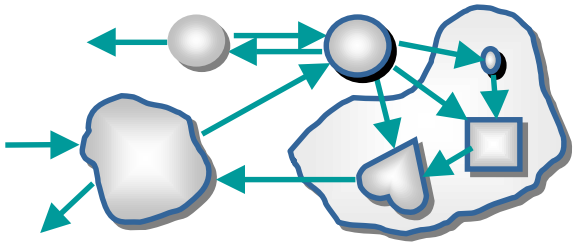

Figure 3: a layered network

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

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

4 The JCSP Binding

JCSP[4] is a Java class library providing a base range of CSP primitives plus a rich set of extensions, some of the latter being experimental at the moment. Also included is a package providing CSP process wrappers giving a channel interface to all Java AWT widgets and graphics operations. It is extensively (javadoc)umented and includes much teaching material. *[Check out, also, the CTJ [12] library.]*

JCSP *(and CTJ)* enables multithreaded systems to be designed, implemented and reasoned about entirely in terms of CSP synchronising primitives (channels, events, etc.) and constructors (parallel, choice, etc.). This allows 20 years of theory, design patterns (with formally proven good properties – such as the absence of race hazards, deadlock, livelock and thread starvation), tools supporting those design patterns, education and experience to be deployed in support of Java multithreaded applications.

4.1 A Process Design Pattern for Java

With JCSP, a process is an instance of a class implementing the `CSProcess` interface:

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

The behavior of a process is defined by the implementation of this `run()` method**.**

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

The structure of a JCSP process should follow the outline:

```
import jcsp.lang.*;
...  other imports

class Example implements CSProcess {

  ...  private shared synchronization
       objects (channels etc.)
  ...  private state information

  ...  public constructors
  ...  public accessors/mutators
       (only used when not running)

  ...  private support methods
       (part of a run)
  ...  public run method
       (the process starts here)

}
```

The pattern of use for these methods is simple and well-disciplined.

The public constructors or mutator methods must install the shared synchronization objects into the private fields. They may also, of course, initialize other private state information.

The public accessor/mutator methods (simple `sets` and `gets`) may be invoked only when this process is not running. They should be the responsibility of a single process only – usually the process that constructed this one.

That constructing process is also responsible for triggering the public `run()` method that kicks this one into life (usually in `Parallel` with some other constructed processes – see Section 4.3). The private support methods are invoked only by each other and by the `run()` method and express the live behavior of this process.

A process instance may have several lives but these must, of course, be consecutive. Different instances of the same process class may, *also of course*, be alive concurrently

When a process is running, it is in sole charge of its private fields. Its thread of control never leaves the process and no foreign threads can enter. No other processes can inspect or interfere with those fields.

Changes of state may be *requested* by other processes (e.g. through channel communication), but this process is at liberty to refuse even to listen to such requests. Both sides must actively cooperate to exchange information – so neither can be surprised when this happens.

Now, consider a JCSP process, `X`, with a private integer field, `count`, and a channel interface through which it communicates with other processes. Suppose the following lines of code occur in its `run()` method (or one of its private support methods):

```
count = 42;
thing.write (something);
```

where `thing` is an output channel. What is the value of `count` after these two lines?

This time we do know. *What-you-see-is-what-you-get*: the answer is `42`. The only way `count` can be changed is if *this* process changes it – and writing something to a channel doesn't do that! Our intuitive understanding of the sequence of instructions has been honored. We have no need to consider what lies on the other side of the `thing` channel – local analysis is sufficient.

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

4.2 Alternation – Choosing between Events

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

JCSP provides a passive mechanism for doing the waiting (there is no active polling for events) and three ways for resolving any offered choice (*arbitrary*, *user-prioritized* and *fair*).

Figure 4 shows a process, `Sample`, that services any of three events (two inputs and one timeout) that may occur. Its parameter, `t`, represents a time interval. Every `t` milliseconds, `sample` must forward (to its `out` channel) the *last* object that arrived (on its `in` channel) during the previous time slice. If nothing has arrived, it

must output `null`. The value of `t` may be reset at any time by a command on its `reset` channel.

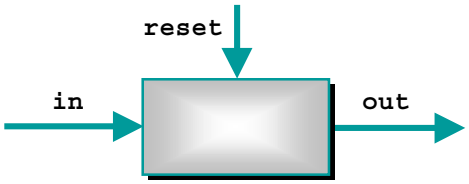

Figure 4: real-time sampler

Here is a JCSP definition of this process:

```
import jcsp.lang.*;

class Sample implements CSProcess {

  private long t;
  private final AltingChannelInput in;
  private final
    AltingChannelInputInt reset;
  private final ChannelOutput out;

  public Sample (long t,
            AltingChannelInput in,
            AltingChannelInputInt reset,
            ChannelOutput out) {
    this.t = t;
    this.in = in;
    this.reset = reset;
    this.out = out;
  }

  ...  public void run ()

}
```

The above code sets up the channel interface and initial time-slice interval. The `in` and `reset` channels are declared for input and `out` for output. The `in` and `out` channels carry Java objects, but the `reset` is reserved for `int`s. The two input channels are explicitly flagged as *Alting* (i.e. useable as operands for a CSP choice).

All the action takes place within the `run()` method, so there is no need for other state fields. Here are the opening declarations of `run()`:

```
public void run () {

  final CSTimer tim = new CSTimer ();

  final Alternative alt =
    new Alternative (
      new Guard[] {reset, tim, in}
    );

  final int RESET = 0, TIM = 1, IN = 2;
```

A `CSTimer` is a JCSP object used for setting timeouts. The `Alternative` is used for waiting and choosing between events (called `Guards` in JCSP) and must be bound to those guards at construction time. Allowable guards are input channels (various kinds), `CSTimers` and `Skips` (which are events that are always pending).

We need to decide how the choice is to be made should more than one of the guards be ready. In this case, using priorities makes sense. The least common event will be the `reset`, so make that the highest priority. The timeout triggers output, which is probably more crucial in this case than accepting a late arriving object on `in`.

When using prioritized choice, the ordering in the `Guard` array bound to the `Alternative` defines those priorities – hence, the sequence we have used. The three named values (`RESET`, `TIM` and `IN`) are indices to this `Guard` sequence and are convenient to have around.

Continuing the `run()` method:

```
  Object sample = null;
  long timeout = tim.read () + t;
  tim.setAlarm (timeout);
```

This declares the variable to hold any samples delivered on the `in` channel – only one is needed since only one sample (the *latest* one) ever needs to be held. Then, the first timeout value is computed (the current time plus the time-slice period) and the alarm call is set (on `tim`).

```
  while (true) {
    switch (alt.priSelect ()) {
      case RESET:
        t = reset.read ();  // assume OK
      break;
      case TIM:
        out.write (sample);
        sample = null;
        timeout += t;
        tim.setAlarm (timeout);
      break;
      case IN:
        sample = in.read ();
      break;
    }
  }

}
```

At the start of the main loop, `alt.priSelect()` waits (consuming no processor cycles) until one or more of its associated events occurs. It, then, makes its choice and returns the index of the chosen one. If that index represents a channel guard, that channel must be `read()`.

The rest of the loop is straightforward serial programming. Incoming samples just overwrite what was previously saved. At the end of a time-slice, the current sample is output and reset to null and the next timeout is computed and set. If a `reset` occurs, the value defines the period of the *next* (and subsequent) time-slices – the current time-slice is not interrupted.

The above code does not check that the `reset` values are sensible (e.g. positive), but that is trivial to add. If we want the `reset` to end the time-slice, read the current time into `timeout` and remove the `break` at the end of the `RESET` case (so that the logic falls through to `TIM`).

4.3 Going Parallel

The code defining a process network is a simple representation of the network diagram.

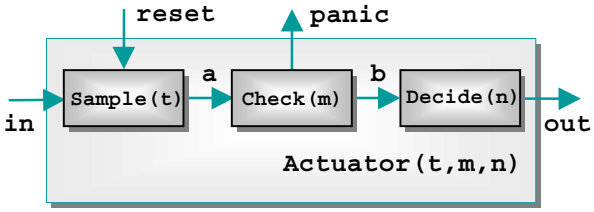

Figure 5: final stage actuator

Figure 5 shows the final stage processing of a real-time control signal. Computed control values are delivered – at varying rates – on the `in` channel to `Actuator`. Hardware control is realized by values sent on the `out` channel and these *must* take place every `t` milliseconds (or control will be lost). Depending on conditions detected externally, the time period for that control may need to be `reset`.

If the external logic fails to deliver a new control value in time, a *best guess* based on the last `n` control outputs must be output. The best guess algorithm will be much simpler and faster than the proper control laws computed by the external logic. If best guesses have to be employed `m` times in a row, a `panic` signal must be generated.

In the above system, `Sample(t)` is responsible for passing on the last properly computed control value in each time-slice, reporting a `null` if none were received and for accepting resets to the time-slice interval.

`Check(m)` just forwards everything it gets, but counts passing `null`s – if it sees `m` in a row, it fires the `panic` signal.

`Decide(n)` passes on non-`null` control values directly, but maintains a buffer of its last `n` outputs. If a `null` arrives, it applies the best guess algorithm and delivers the result.

Here is the code for this network:

```
class Actuator implements CSProcess {

  ...  private state (t, m and n)

  ...  private interface channels
       (in, reset, panic and out)

  ...  public constructor
       (assign t, m, n, in, reset, panic
        and out parameters to the above)

  public void run () {

    final One2OneChannel a =
      new One2OneChannel ();

    final One2OneChannel b =
      new One2OneChannel ();

    new Parallel (
      new CSProcess[] {
        new Sample (t, in, reset, a),
        new Check (m, a, panic, b),
        new Decide (n, b, out)
      }
    ).run ();

  }
```

The hidden parts are the same as their equivalents in `Sample` – except that there are two extra state fields and one more channel.

The interesting part is the `run()` method. This declares the two internal wires (`a` and `b`) needed to make up the circuit. These are *actual* channel objects (as opposed to the *formal* channel parameters, which are known to this process only through Java interfaces). Instances of this process will be given actual channel arguments when they are constructed. In turn, those instances pass on the channels they are given, and/or the new ones they make, to the sub-processes they construct.

The JCSP `Parallel` constructor takes an array of (sub-)processes and returns a process, which is then run. That run is the *parallel composition* of the given processes. It terminates when, and only when, all its given processes terminate.

5 Conclusions

This paper has outlined the basic primitives for process-oriented design: processes, synchronizing channel communication, alternation and parallel composition. It has demonstrated the *WYSIWYG* nature of CSP design whereby each process can be considered individually, computing on its own data and interacting with its environment through synchronizing I/O devices (e.g. channels). Leaf processes in the network hierarchy are traditional *serial* programs, where all our past skills and intuition can safely be applied. The new skills for concurrency sit happily alongside those earlier ones, with no cross-interference.

The JCSP library contains much more than the set of classes introduced above to illustrate CSP design for Java. Several varieties of channel are provided (e.g. *buffered*, *any-to-any* and *variant calls*) plus other kinds of synchronization (e.g. *barrier*, *bucket* and *CREW*), but all have well-defined CSP characterization. JCSP also provides a complete set of AWT classes converted into active processes with channel interfaces.

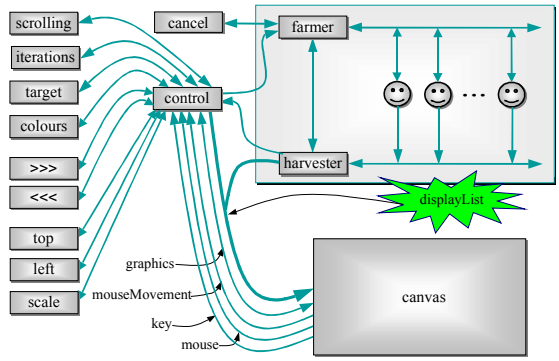

Figure 6: a GUI/graphics network

Figure 6 shows an application making use of these facilities – only the *smiley* workers and the `control`, `farmer` and `harvester` processes are specially written. It contains a classic *process farm* for generating graphics images with GUI control over that generation. This (Mandelbrot) example is one of the demonstrations included in the JCSP release. On a multiprocessor (SMP) architecture, the process farm yields almost linear speed-up for image computation.

Not addressed in this paper are the avoidance of race hazards and deadlock. Hazards arise from unsynchronized access to shared resources (e.g. when object *references* are passed between processes). However, because all processes must actively be involved in such distribution, design rules are not hard to devise to ensure safety. A rich set of design rules (e.g. [13-16]) also exists that have formally proven guarantees against deadlock and livelock errors. Future work will look to provide design tools to automate and/or police these rules as well as to combine this work with developments in **occam** (e.g. CSP kernels with *nanosecond* management overheads [17]).

6 References

[1] C.A.R.Hoare. *Communicating Sequential Processes.* CACM, 21-8, pp. 666-677, August 1978.

[2] C.A.R.Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[3] A.W.Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, ISBN 0-13-674409-5, 1997.

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

[5] P.H.Welch, *Java Threads in the Light of occam/CSP*. In 'Architectures, Languages and Patterns for Parallel and Distributed Applications', WoTUG-21, pp. 259-284, IOS Press (Amsterdam), ISBN 90 5199 391 9, April 1998.

[6] P.H.Welch. *Parallel and Distributed Computing in Education*. In J.Palma et al. 'VECPAR'98', Lecture Notes in Computer Sciince, vol. 1573, Springer-Verlag, June 1998.

[7] G.S.Stiles, A.Bakkers, G.H.Hilderink and P.H.Welch. *Safe and Verifiable Design of Concurrent Programs*. In 'Proceedings of the 3rd. International Conference on Software Engineering and Applications', pp 20-26, IASTD, Oct. 1999.

[8] D.Lea. *Concurrent Programming in Java (Second Edition): Design Principles and Patterns*. The Java Series, Addison-Wesley, section 4.5, 1999.

[9] J.M.R.Martin and P.H.Welch. *A CSP model for Java Multithreading*. In 'Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000)', IEEE Computer Society Press, June 2000 *(to appear)*.

[10] P.Brinch-Hansen. *Java's Insecure Parallelism*. ACM SIGPLAN Notices, 34, 4, pp. 38-45, April 1999.

[11] Formal Systems (Europe) Ltd. *Failures-Divergence-Refinement: FDR2 Manual*, http://www.formal.demon.co.uk/FDR2.html , 1997.

[12] G.H.Hilderink. *The CTJ (Communicating Threads for Java) home page*. http://www.rt.el.utwente.nl/javapp/

[13] P.H.Welch, G.R.R.Justo and C.Wilcock. *High-Level Paradigms for Deadlock-Free High-Performance Systems*. In 'Transputer Applications and Systems 1993', pp. 981-1004, IOS Press (Amsterdam), ISBN 90 5199 140 1, 1993.

[14] J.M.R.Martin and P.H.Welch. A Design Strategy for Deadlock-Free Concurrent Systems. Transputer Communications 3(4), pp. 215-232, John Wiley & Sons, ISSN 1070 454 X, October 1996.

[15] J.M.R.Martin and S.A.Jassim. *A Tool for Proving Deadlock Freedom*. In 'Parallel Programming and Java', WoTUG-20, pp1-16, IOS Press (Amsterdam), ISBN 90 5199 336 6, 1997.

[16] J.M.R.Martin, *A Tool for Checking the CSP **sat** Property*. The Computer Journal, Vol. 43, No. 1, 2000.

[17] P.H.Welch, J.Moores, F.R.Barnes and D.C.Wood. *The KRoC Home Page*. http://www.cs.ukc.ac.uk/projects/ofa/kroc/, 2000.