Published in Real-Time Magazine 98/1 pp.30-35

A new Java Thread model for concurrent programming of real-time systems.

Gerald Hilderink
Jan Broenink
André Bakkers
(G.H.Hilderink, J.F.Broenink, A.W.P.Bakkers)@el.utwente.nl
University of Twente, dept. EE, Control Laboratory,
P.O.Box 217, 7500 AE Enschede, The Netherlands

Abstract. The Java™ Virtual Machine (JVM) provides a high degree of platform independence, but being an interpreter, Java has a poor system performance. New compiler techniques and Java processors will gradually improve the performance of Java, but despite these developments, Java is still *far* from real-time.
We propose the *Communicating Java Threads* (CJT) model, which eliminates several shortcomings, such as Java's non-deterministic behavior, Java's monitor weakness, and lack of reactiveness for real-time and embedded systems. CJT is based on CSP providing channels, composition constructs, and scheduling of processes. The CJT Java class library, which provides all necessary classes on top of Java, is readily available to interested users. The main power of the method is that it integrates these features in a natural way and in no way conflicts with other paradigms, such as PersonalJava™, EmbeddedJava™, JavaBeans™, CORBA, and PVM. All activities, reported in this paper, are carried out as part of the JavaPP project, see http://rt.el.utwente.nl/javapp.

1. Introduction

Real-Time system design traditionally is the domain of experienced specialists, who were faced with a variety of custom kernels, non-standard and mostly low-level languages and vendor specific I/O device interfaces. The proposed new Java Thread model is to remedy this situation. The proposed method addresses four serious deficiencies of current methods for the design of real-time embedded systems: security, performance, portability, and development time. *Security* is compromised by the complexity and ill-definition of the interfaces between current programming languages and real-time operating-system/kernels. These semantics cannot be formalised in ways that guarantee the safety of systems. *Performance* suffers because engineers have to accept whatever levels of abstraction are provided by the real-time kernel. If these are too high or too low, unnecessary overheads are incurred. *Portability* problems arise because of the vendor specific nature of current tools. The *development time* of real-time embedded systems is hindered by the lack of development tools to bridge the gap between design, expressed as a collection of communicating processes, and implementation on serial hardware.
We introduce the use of CSP (Communicating Sequential Processes) channels [3] in Java providing a communication harness to interconnect processes. The required software is available as a Java class library and may be downloaded from our web server (http://rt.el.utwente.nl/javapp).
The use of Java for the design of real-time systems may rightfully be frowned upon by real-time system designers. Because of a number of weaknesses that Java exhibits in this area.

The major weaknesses lie in the Java's, dynamic memory management, the threads model, and the lack of immediate responsiveness or reactiveness to its environment. This has been summarized in the following three points.

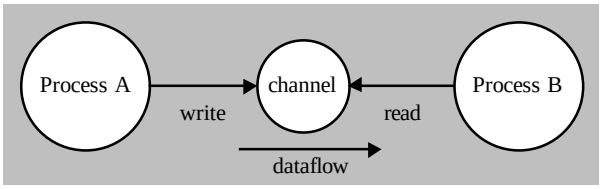
1. Java contains several dynamic components that may jeopardize the deadlines of processes. In Java, the cloning concept (clonable and serializable objects) requires dynamic memory allocation and is dependent on garbage collection. Other components, such as the class loader and the network classes also use dynamic memory management. Deterministic behavior is most important for developing software for real-time and embedded systems. Generally, static memory allocation without garbage collection is used. Evading the garbage collector is hard and several groups are working on deterministic garbage collectors.
2. The monitor concept in Java (`synchronized-wait-notify`) works statistically correct, but the risk of *starvation* and *livelock*, or in the worst case *deadlock*, may occur when adding specific time constraints. According to Welch [5] the Java monitor is inefficient and dangerous to use. In the current implementation of the Java monitor, threads will queue up at the end of a single `synchronized` queue when the monitor has already been claimed by another thread. When many threads cyclically claim the monitor and on bad luck with timing, it may happen that a particular thread never gets off the queue. The results will be infinite starvation and livelock [5]. A correct implementation must be provided by a real-time operating system (RTOS).
3. The presence of a real-time scheduler – as part of a RTOS – doesn't necessarily make a real-time system. A real-time system should also be reactive on incoming stimuli, such as an alarm sensor or a sample clock. Java has no facilities for interrupt handling, therefore, this must be implemented in the RTOS. A lot of things must be solved at operating system level. Java on the other hand is more suitable in that it is naturally component oriented.

The *Communicating Java Threads* class library [1] we have developed, solves the above mentioned problems by means of formal methods. The total concept [2] is derived from the mathematical process algebra CSP [3], programming language occam [4], and the transputer. These technologies contain formal techniques that have been proven in the real-time and embedded system community for many years. With this formal approach and the use of concise rules one can prevent race-hazards, livelock, starvation, and deadlock during design and implementation. The concept offers the possibility to develop a number of design patterns. With the use of formal methods one can guarantee the behavior of the system. In other words, formal methods are essential to the development of reliable systems. We restrict ourselves to utilize the CSP concept at the above mentioned real-time problems.

In this paper, in section 2 the CSP channels are introduced. The impact of the use of channels on the plug-and-play concept is described in section 3. In section 4 the relation between the channel concept and real-time behavior is described, while in section 5 our scheduler methods are discussed. In section 6 conclusions are given towards the use of CSP channels in real-time and embedded systems.

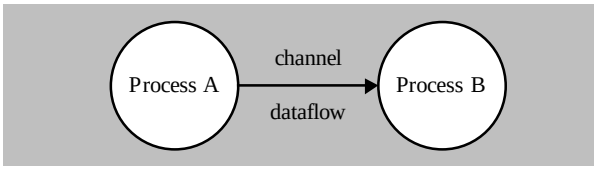
2. Programming with channels

The CSP concept offers a different parallel model than the Java Thread model. The CSP model defines so-called channels that synchronize processes through communication between these processes.
With the term *Java channels* we restrict ourselves to CSP channels – also called *communication events*.
The Java channels are intermediate objects shared by active objects – threads or processes – in which they communicate with each other. Channels are one-way, initially unbuffered, and fully synchronized. This is illustrated in figure 1. Active objects can only read or write on channels. Communication occurs when both processes are ready to communicate. Processes get blocked until communication is ready. Synchronization, scheduling, and the physical data transfer are encapsulated into the channel. The result is that the programmer will be freed from complicated synchronization and scheduling constructs.



Figuur 1. Object oriented channel communication.

The channel model reduces the gap between concurrent design models, such as data-flow models, and the implementation. Figure 2 represents a data-flow at design level, whereas figure 1 represents the implementation.



Figuur 2. Dataflow or process oriented.

In data-flow diagrams, an arrow corresponds to a channel and a circle represents a process. The mapping of a dataflow diagram to code is straightforward; the one-way directed arrows represent the input/output *interfaces* of the circles whereas input or output channels define the interfaces of the processes.

Listing 1 illustrates this by mapping the design of figure 2 into code.

```
1.  import cjt.*;

2.  public class Main
3.  {
4.    public static void main(String[] args)
5.    {
6.      Channel channel = new Channel();

7.      ProcessA pa = new ProcessA(channel);
8.      ProcessB pb = new ProcessB(channel);
9.    }
10. }
```

Listing 1. Main program

The `main()` method acts as a so called *configurer*, which typically declares channels and processes once and subsequently terminates. This method does not contain a common loop construct. Instead, less complex loops are build in the processes.

The `Main` class represents a concurrent program for one processor. It is trivial to split up the `Main` class into several configurer classes for each processor. Important is that the processes will stay intact and channels possess the knowledge of the media between the processors (see section 3).

In listings 2 and 3 ProcessA (producer process) and ProcessB (consumer process) are given. ProcessA produces 10000 integer incrementing numbers starting from zero. ProcessB consumes these 10000 numbers and prints them onto the screen.

```
1.  import cjt.*;
2.  import java.io.IOException;

3.  public class ProcessA extends ProcessObject
4.  {
5.    ChannelOutput channel;

6.    public ProcessA(ChannelOutput out)
7.    {
8.      channel = out;
9.      start();
10.   }

11.   public void run()
12.   {
13.     IntegerObject object = new
    IntegerObject();
14.     SeqComposition seq = new SeqComposition();

15.     try
16.     {
17.       while(object.value < 10000)
18.       {
19.         object.value++;
20.         channel.write(seq, object);
21.       }
22.     } catch (IOException e) { }
23.       catch (InterruptedException e) { }
24.   }
25. }
```

Listing 2. Producer ProcessA.

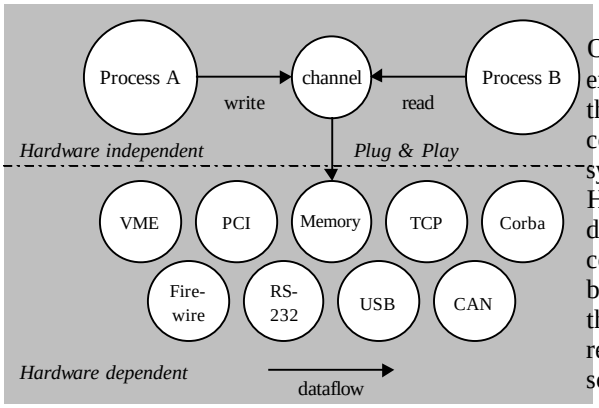
```
1.  import cjt.*;
2.  import java.io.IOException;

3.  public class ProcessB extends ProcessObject
4.  {
5.    ChannelInput channel;

6.    public ProcessB(ChannelInput in)
7.    {
8.      channel = in;
9.      start();
10.   }

11.   public void run()
12.   {
13.     IntegerObject object = new IntegerObject();
14.     SeqComposition seq = new SeqComposition();

15.     try
16.     {
17.       while(object.value < 10000)
18.       {
19.         channel.read(seq, object);
20.         System.out.println(object.value);
21.       }
22.     } catch (IOException e) { }
23.       catch (InterruptedException e) { }
24.   }
25. }
```

Listing 3. Consumer ProcessB.

The `read()` en `write()` methods contain two arguments `seq` and `object`. The object `seq` is of class type `SeqComposition`, which indicates sequential behavior of the `read()` and `write()` methods. There also exists a `ParComposition` en `AltComposition` for respectively parallel and alternative behavior. These composition constructs are described in the reference manual, which can be found at our Web site.
Channels are thread-safe for multiple readers and writers. Multiple consumer and producer processes may share the same channel. The channel also performs scheduling between processes of different priority. The priority of process can be determined by means of the priority of communication. Naturally, one-to-one and many-to-one relations can be realized. A one-to-many (broadcasting) relation needs a separate design pattern.

3. Plug-and-Play

The channel concept in Java goes beyond communication only. The core Java development kit has no framework for direct hardware support. Java does support native languages, such as C/C++, through the JNI (Java Native Interface). The more code that will be written with JNI the less Java will be portable on other platforms. The channel concept defines an abstract way to control devices and confine hardware dependent code to one place only. This approach enlarges the reusability, extensibility, and maintainability in an object oriented manner.
Channels between processes on one processor will use a shared memory driver and channels between processes on different processors will use a peripheral driver. The result is that processes will always be hardware independent. There will be a clear separation between hardware dependent and hardware independent code as illustrated by figure 3.



Figuur 3. Plug and Play framework for devices.

To avoid developing special channels for each peripheral, a device driver framework is developed. The device drivers, we call them *link drivers*, are hardware dependent objects that can be plugged into the channel. The channel object will deal with synchronization and scheduling and the link driver will be responsible for the data transfer. Channel objects are hardware independent. As a result, the link driver will be freed from the tasks of synchronization and scheduling, therefore programming link drivers will become easier.

The `read()` and `write()` methods are, when permitted by the synchronization mechanism of the channel object, delegated to the link driver. Figure 4 shows communication between two processes on one processor, whereas figure 5 shows communication between two systems.

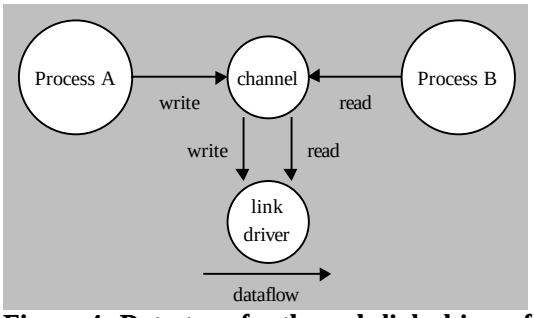


Figure 4. Data transfer through link drivers for uni-processor systems.

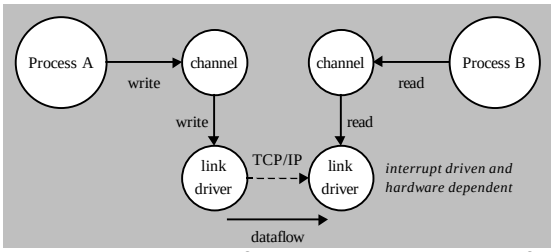


Figure 5. Data transfer through link drivers for multi-processor systems.

Declaring a channel with a link driver is illustrated by the following code:

```
Channel chan = new Channel(new MyLinkDriver());
```

On our URL http://rt.el.utwente.nl/javapp we have an example TCP/IP link driver for communication across the Internet. This example also shows how a single concurrent program can be split up for distributed systems.
Hardware dependent objects can be found at the declaration of channels, i.e. at the top-level of the configurer. It will be easier for the programmer to browse and to maintain the program, without changing the processes. Again, this concept increases the reusability, extendibility, and maintainability of the software.

4. Real-time aspects

Timing is one the most critical aspects of modern real-time systems. From the requirement point of view, we are only concerned with *external* timing [7]. The users are concerned only that the system will respond overall to a certain stimulus within certain time constraints. Whether the response was achieved by a background task or foreground task, how it was scheduled relative to other tasks, what the internal port-to-port timing was, and what kind of executive controller was needed to achieve it, are issues that do not concern the system designers. From the same point of view, timing is related only to the signals of the system interface as indicated by the context diagram. The context diagram is a data-flow diagram at the top-level by which signals flow between the system and the peripherals or terminators. In the context diagram, the arrows are also communication channels where communication takes place at specified times. This makes channels important for real-time systems.
The channel concept offers a solution to the realization of real time requirements as follows:

1. The non-deterministic behavior of Java that is caused by cloning objects or object serialization, together with garbage, collection can be avoided when using channels. Channels copy the contents of the source object to the destination object when communication is ready. Therefore objects can efficiently be reused and the process behavior will be deterministic. However, this copying concept does not conflict with the cloning and serialization concepts of Java, which can be used as well. However it does not require the need of garbage collection.
2. Special link drivers may extend the scheduling behavior of the channel. A link driver consisting of a one-place buffer may alleviate the priority inversion problem combined with a rate monotonic priority scheduler. Despite general believe this type of scheduler may be used to the full 100% CPU utilization [6].
3. Interrupt handling in a channel philosophy becomes the scheduling of a respective process at the required priority. This is implemented as the placement of that particular process in its respective active queue.
4. The channel can be fully optimized for processes of equal priorities and also for processes of

different priorities separately. The processes will not notice any changes between different channels, because of its unique interface definition.

From the above, it can be concluded that the programmer can safely concentrate on the use of channels whereas inside the channels the embedded scheduler takes care of the proper scheduling without any user intervention. The implications of the embedded scheduler are therefor considered next.

5. Embedded scheduler

Thread scheduling belongs to the domain of the application that executes its tasks in parallel. So, a scheduler is not necessarily part of the operating system. A scheduler as an operating system resource is all right for multitasking, but we can generalize this concept. Each concurrent program may have its own embedded scheduler that schedules its parallel tasks. An operating system may be a concurrent program as well. In other words, there may be more than one scheduler running in a system; a scheduler may schedule another scheduler and so on. In such a way, different concurrent programs can be scheduled in it's own way. An embedded scheduler cuts a thread into multiple threads by means of task switching (or context switching).

This approach corresponds to the objectives of object orientation. A single program is an active object, i.e. an object with a life of its own, with its behavior encapsulated within. A concurrent program contains multiple active objects, which must be scheduled. Thus, concurrent programs need an embedded scheduler that takes care of multithreading. An embedded scheduler is also an object that is part-of the total program. There are several benefits to this approach:

1. The scheduler object can be included when one is needed. When more than one schedulers of the same type is needed only one code segment resides in memory.
2. Different types of schedulers can be used. The scheduling behavior can be nested such that logistic and real-time policies can be mixed.
3. The Java Virtual Machine may be simplified by leaving out the scheduler part. This makes the JVM more compact and better portable. The scheduler classes can be treated as object oriented concepts and are therefore better maintainable, extensible and reusable.
4. The open interface of this approach permits other design patterns for concurrent programming and scheduling policies.

With the `wait()`, `notify()`, and `synchronized()` methods one can write robust programs according to certain design patterns [9]. One should follow to these patterns closely otherwise this great extend of freedom may turn into a source of errors.

The channel concept comprehends several design patterns, which are very related to each other. These patterns deal with avoiding deadlock, starvation and livelock at a more abstract level than dealing with hazardous synchronization concepts. Together with the embedded scheduler concept and the link driver framework we can deal with real-time constrains at a more abstract level through design patterns based on the channel concept.

6. Conclusions

The use of CSP channels in real-time system design offers a unified framework that clears the programmer from complicated and unnecessary programming tasks such as thread programming and scheduling. The proposed method allows for deadlock and starvation checks. The notion of priority should be considered a property of communication between processes rather than of processes by itself.

The resulting programs are easy to read and maintain. The resulting code is as fast or as slow as equivalent well-written Java code. Experience from the past has learned that CSP channels may be designed with lightning speed. There is sufficient room for performance improvement and this should be undertaken in parallel to the activities to make Java more suitable for real-time programming in general.

The Java channels as introduced do not require garbage collection.

References

[1] G.H. Hilderink, "Communicating Java Threads - Reference Manual", Proceedings of WoTUG-20 conference *Parallel Programming and Java*, IOS Press, 1997, pp.283-324.

[2] G.H. Hilderink et al., "Communicating Java Threads", Proceedings of WoTUG-20 conference *Parallel Programming and Java*, IOS Press, 1997, pp.48-76.

[3] C.A.R. Hoare, "Communicating Sequential processes", *Communications of the ACM*, Aug. 1978, pp. 666-677.

[4] INMOS Ltd., "Occam2 Reference Manual", *Prentice-Hall* (ISBN 0-13-629312-3), 1987.

[5] P. H. Welch, *"Java Threads in the Light of CSP"*, *Proceedings NLUUG Conference*, Ede, The Netherlands, 20 Nov. 1997. see: http://www.nluug.nl/nluug/nj97/progboekje/lezing10.html.

[6] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", *Journal of the ACM*, Volume 20, Number 1 (January 1997), pp. 46-61.

[7] D.J. Hatley and I.A. Pirbhai, "Strategies for Rel-Time System Specification", Dorset House

Publishing Co. (ISBN 0-932633-11-0), New York, 1988.

[8] A. Bakkers, J. Sunter and E. Ploeg, "Automatic generation of scheduling and communication code in real-time parallel programs", *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers & Tools for Real-Time Systems*, La Jolla, California, June 21-22, 1995

[9] D. Lea, "Concurrent Programming in Java: Design Principles and Patterns", Addison-Wesley Publishing Co. (ISBN 0-201-69581-2), Massachusetts, 1997.

About the authors

Gerald H. Hilderink is currently a Ph.D. student in computer engineering at the Control Laboratory of the University of Twente, The Netherlands. His main research interest is the development of a reliable foundation using formal methods for real-time and embedded system design. He is interested in proposing a new parallel model for real-time and embedded systems in Java.

Jan F. Broenink received his Ph.D. in Electrical Engineering in 1990 from the University of Twente. His Ph.D. research was in the design of computer facilities for modeling and simulation of physical systems using bond graphs. He is presently Assistant Professor at the Control Laboratory of the Department of Electrical Engineering of the University of Twente, where he the project leader software tools development. His research interests include development of computer tools for modeling, simulation and implementation of embedded control systems and robotics.

André W.P. Bakkers is an appointed professor at the Dutch Open University in the field of information technology with a special assignment in the area of real time and parallel systems. Since 1977 he worked in the Control Laboratory of the University of Twente as a Senior Lecturer with the control systems and computer-engineering group of the electrical engineering faculty of the University of Twente. His present research covers the realization of real-time control systems using parallel processing.

Correspondence address

University of Twente
Prof.ir. A.W.P. Bakkers
Faculty of Electrical Engineering
Control laboratory
7500 AE Enschede
The Netherlands
tel. +31-53-482606
fax. +31-53-4892223