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

A Distributed Real-Time Java System Based on CSP

André Bakkers, Gerald Hilderink and Jan Broenink
Control Laboratory, University of Twente
P.O. Box 217, NL-7500 AE Enschede
bks@el.utwente.nl

Abstract: Real-time embedded systems in general require a reliability that is orders of magnitude higher than what is presently obtainable with state of the art C programs. The reason for the poor reliability of present day software is the unavailability of a formalism to design sequential C programs.
The use of the CSP channel concept not only provides a formal base for inherently concurrent real-time embedded system design it also adds a parallel dimension to object oriented programming that is easily understood by programmers.
The CSP channels as implemented in Java replaces the hazardous use of multi threaded programming with an unambiguous design concept that is easy to reason about. Multi threaded programming is completely removed from the programmer who is merely required to program small sequential tasks that communicate with each other via these CSP channels. The channel concept that has been implemented in Java deals with single- and multi processor environments and also takes care of the real-time priority scheduling requirements. For this, the notion of priority and scheduling have been carefully examined and as a result it was reasoned that both priority and scheduling code should be attached to the communicating channels rather than to the processes. Moreover in the proposed system, the notion of scheduling is no longer connected to the operating system but has become part of the application instead. One has to get used to the idea that many schedulers may be running in different parts of a program. The software implementation of the Java channel class may be obtained through: http://www.rt.el.utwente.nl/javapp.

1. Introduction

The design of real-time embedded systems traditionally is the domain of distributed heterogeneous multi processing. It should be recognised that there is no formal method to perform the design of sequential embedded software. As a consequence the resulting software may be working but exhaustive testing may soon be impractical due to time limitations.

The concept of CSP - 'Communicating Sequential Processes' as introduced by Hoare [1] and more recently brought up to date by Roscoe [2], constitutes a formalism that may be used to design distributed real-time Java programs. Whereas real-time embedded systems, when for instance analysed by suitable CASE tools, exhibit lots of parallelism the use of CSP channels implicitly renders a formal design method for which tools exist to proof correctness of an implementation. The CSP channels as implemented in Java should be used instead of the traditional *method calls* of concurrent tasks. The reason for this is that these method calls form the basis of the sequential analysis of object oriented programming. When the method calls are replaced by CSP channels this sequential limitation is removed from object orientation and makes object orientation truly concurrent. As a result fully parallel data flow diagrams may be implemented using these channels. The question one may ask is whether this channel concept holds ground for real-time aspects like scheduling and priority handling. In the following section of this paper the notion of priority and scheduling will be defined and it will be shown that the channel as implemented fully supports the real-time aspects of embedded system design including the distributed environment resulting in a provable correct implementation of the system.

2. What are CSP channels?

The basic idea of CSP as defined by Hoare is that parallel or concurrent processes may work together by synchronising on the respective inputs and outputs of these processes. The communication as proposed by Hoare can only occur if process A indicates that it is ready to output to process B and at the same time process B indicates that it is ready to receive input from A. As long as only one of these conditions is true, the associated process will be put on hold (de-scheduled) until the other process is ready as well. CSP uses ‘channels’ to realise communication between processes or tasks exclusively. Moreover, CSP specifies other constructs that determine the sequence of execution of the different processes or tasks. These constructs are: SEQ for sequential, PAR for parallel and ALT for alternative execution of processes. The SEQ construct determines that processes listed under this construct are executed in sequential order. The PAR construct determines that processes listed under this construct are executed concurrently i.e. in parallel. The PAR construct is completed after the execution of all processes listed under the PAR construct. Synchronisation and scheduling of these processes is controlled by the channels. The ALT construct consists of guards that in turn each guard a process. As soon as a guard becomes ready it is executed followed by the execution of the guarded process, this completes the execution of the ALT.

```
SEQ                     PAR                    ALT
  process-1               process-1              (guard-1)
  process-2               process-2                (guarded process-1)
   ...                    ...                    (guard-2)
   process-n              process-n                (guarded process-2)
```

The processes in the above constructs communicate with each other by means of CSP channels as defined before.

2.1 Java Channels

When we use the term Java channels we imply the use of CSP channels exclusively. The Java channels are passive intermediate objects shared by active objects – threads or processes – in which they communicate with each other. Channels are one-way, initially unbuffered, and fully synchronised. This is illustrated in figure 1. Active objects may only read or write on channels. Communication occurs when both processes are ready to communicate. Processes get blocked until communication is ready. Synchronisation, scheduling, and the physical data transfer are encapsulated in the channel. The result is that the programmer is freed from complicated synchronisation and scheduling constructs.

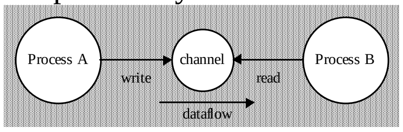

Figure 1 Object oriented channel communication

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. In data-flow diagrams, an arrow corresponds to a channel and a circle represents a process.

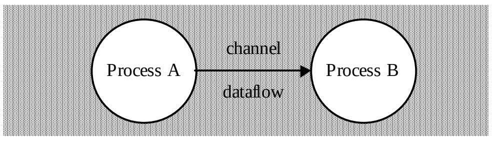

Figure 2 Data flow or process oriented

The mapping of a data flow 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.

```
import csp.lang.*;
public class Main
{
  public static void main(String[] args)
  {
    Channel_of_Integer channel = new Channel_of_Integer();
    Parallel par = new Parallel(new Process[]
    {
      new ProcessA(channel),
      new ProcessB(channel)
    });
    par.run();
  }
}
```

Listing 1 Main Program

The main() method acts as a so called *network builder* or *configurer*, which typically declares channels and processes and executes the processes in parallel. 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 10,000 integer incrementing numbers starting from zero. ProcessB consumes these 10,000 numbers and prints them onto the screen.

```
import csp.lang.*;
import csp.lang.Integer;
import csp.lang.Process;
import csp.io.IOException;
public class ProcessA implements Process
{
  ChannelOutput_of_Integer channel;
  public ProcessA(ChannelOutput_of_Integer out)
  {
    channel = out;
  }
  public void run()
  {
    Integer object = new Integer();
    try
    {
      while(object.value < 10000)
      {
        object.value++;
        channel.write(object);
      }
    } catch (IOException e) { }
  }
}
import csp.lang.*;
```

Listing 2 Producer ProcessA

```
import csp.lang.Integer;
import csp.lang.Process;
import csp.lang.System;
import csp.io.IOException;

public class ProcessB implements Process
{
  ChannelInput_of_Integer channel;

  public ProcessB(ChannelInput_of_Integer in)
  {
    channel = in;
  }
  public void run()
  {
    Integer object = new Integer();
    try
    {
      while(object.value < 10000)
      {
        channel.read(object);
        System.out.println(object.value);
      }
    } catch (IOException e) { }
  }
}
```

Listing 3 Consumer ProcessB

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 is handled in the communication channel as well. Naturally, one-to-one and many-to-one relations can be realized. A one-to-many (broadcasting) relation needs a separate design pattern.

The Java channels have been implemented as channel classes for Java both by the University of Twente and the University of Kent. The taxonomy of the JavaPP library development is illustrated in figure 3.

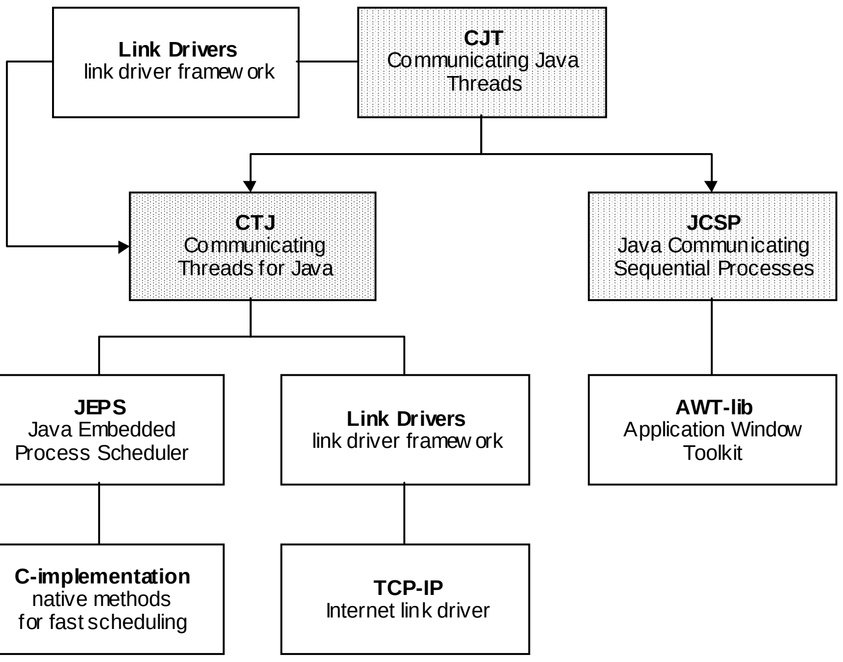
Link Drivers
link driver framework
CJT
Communicating Java Threads
CTJ
Communicating Threads for Java
JCSP
Java Communicating Sequential Processes
JEPS
Java Embedded Process Scheduler
Link Drivers
link driver framework
AWT-lib
Application Window Toolkit
C-implementation
native methods for fast scheduling
TCP-IP
Internet link driver

Figure 3 Taxonomy of the JavaPP library development

The taxonomy of the different JavaPP libraries is illustrated in figure 3. The grey boxes represent the major libraries i.e. CJT, CTJ and JCSP. The original CJT - ‘Communicating Java Threads’ class library as published in Hilderink[3] is based on the so-called link driver concept. This link driver concept makes channels suitable for multi-processor use. Background information on the development of CJT may be found in Welch[4]. The CJT class library was subsequently used by Austin[5] to develop a new Java/CSP library called JCSP - ‘Java Communicating Sequential Processes’, incorporating a proper interface to AWT - ‘Application Window Toolkit’.

At Twente, the CJT library together with the link driver concept has been converted into the CTJ - ‘Communicating Threads for Java’ library implementing the Java embedded process scheduler - JEPS, by Hendriks[6]. The name change from CJT to CTJ was ‘suggested’ by Sun with the 100% pure Java certification of the CJT class library. The CTJ manual is described in Hilderink[7]. The CTJ library was subsequently used as a design pattern to develop a ‘C’ version of CSP channels using the link driver concept together with the embedded process scheduler. The resulting ‘C’ code performed two orders of magnitude faster than the original Java program.

The Twente approach is based on the Channel class using the link driver concept. This Link Driver concept will therefore be clarified in the following section.

3. The Link Driver Concept

The channel concept in Java goes beyond just communication. 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 use a shared memory driver and channels between processes on different processors use a peripheral driver. The result is that processes are always hardware independent. There will be a clear separation between hardware dependent and hardware independent code as illustrated in figure 4.

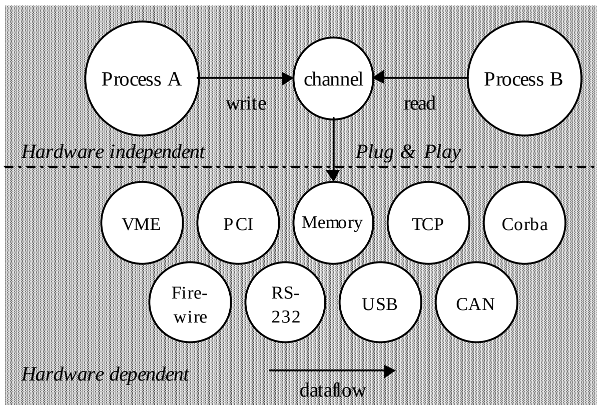

Figure 4 Plug and Play framework for devices

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 becomes easier.

The `read()` and `write()` methods are, when permitted by the synchronization mechanism of the channel object, delegated to the link driver. Figure 5 shows communication between two processes on one processor, whereas figure 6 shows communication between two systems.

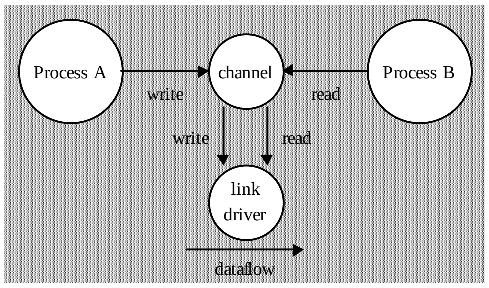

Figure 5 Data transfer through link drivers for uni-processor systems

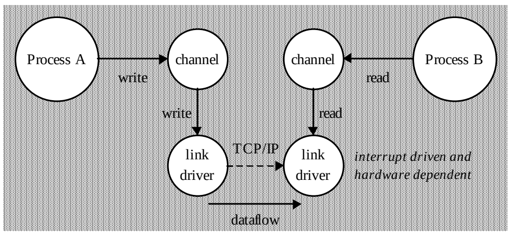

Figure 6 Data transfer through link drivers for multi-processor systems

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

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

Our URL http://rt.el.utwente.nl/javapp contains 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 network builder c.f. listing 1. 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. The notion of Priority

When we think of priority we associate it almost always with processes. A high priority process that is actively executing on a CPU, however, could care less about its priority! That information is totally irrelevant, until the point when this process has to communicate with another process with a lower priority. So actually it is the *communication channel* that is burdened with the task to resolve the differences in priority. This results in the well known priority inversion problem[8].

The priority inversion problem may be resolved by implementing the proper algorithm in a priority inversion link driver. The resulting channel may then be used to handle the difference in priority. This approach calls for the use of Java channels to implement priority scheduling.

4.1 First choice: scheduling algorithm

The first choice one has to make is which priority scheduling algorithm has to be selected. Basically there are two alternatives[9] i.e. the Rate Monotonic (RM) and the Deadline Driven (DD) algorithm. The RM algorithm, which uses a fixed priority list, has in its unrestricted form the disadvantage that the CPU utilisation should be kept below 70% in order to allow for wrong scheduling choices due to this fixed priority list. The DD algorithm requires continuous sorting of its priority list which makes it less attractive in a real time environment.

In real-time embedded systems, however, there is a control-engineering-theoretical need to have measurements coincide with each other. The easiest way to do this is to derive timing sequences from a divider chain fed by crystal controlled oscillator. Thus the ratio of the maximum (mth) sampling time and the ith sampling time becomes an integer as:

$$2^{m-i} \text{ is an integer}$$

The consequence of this is that the CPU utilization no longer has to satisfy the 70% restriction. This somewhat restricted form of the RM algorithm may therefore be operated to the full 100% CPU utilization. This makes the RM algorithm a logical choice for implementation[10].

Having selected the scheduling algorithm, the best way to implement this algorithm is the PRI PAR construct. This construct acts like a PAR construct with the additional requirement that the sequence of the processes listed in the PAR construct also gives the descending order of the priority. Clearly there should be no limit to the number of processes under the PRI PAR construct and even nesting of PRI PAR's is allowed as in:

```
PRI PAR
  process-1
  process-2
  PRI PAR
    process-3
    process-4
  process-5
  ...
```

In our implementation the PRI PAR may have a maximum of 7 processes, because every PRI PAR is being decoded in one byte with one bit per priority. The eighth bit is used for idle tasks, skip tasks or garbage collection. The latter has not been implemented yet because it requires a preemtive garbage collector.

This type of scheduling has an great impact on the use of the scheduler. It requires a shift from the traditional philosophy where the real-time kernel is part of the operating system to the notion that the scheduler should be part of the application. With the acceptance of this shift the implementation of the above PRI PAR becomes compositional. However, the consequence is also that we can no longer use the JVM scheduler, because that would cause the JVM scheduler to clash with the application scheduler. It means that the resulting code is more Java as a language then as a virtual machine. The advantage of this is that the resulting code may be considered more a design pattern[9] and as such may be used to program an identical system in other languages.

Besides, at the present time, the execution of the Java version is too slow for real time systems anyway and should be accelerated at least three orders of magnitude in order to be representable of present day processor speeds. We have transformed the Java design pattern to a 'C' version and have realized an improvement of two orders of magnitude in execution speed although still a factor 10 too slow. The embedded scheduler concept will be further explained in the next section.

4.2 Execution of the PAR versus the PRI-PAR

The execution of processes under a PAR and under a PRI-PAR may be illustrated with the following example. Consider a network of six processes communicating via channels as shown in figure 7. The CSP-like notation used is borrowed from Lawrence[12].

Process PROGRAM1 is composition of a set of parallel processes with equal priorities. This is illustrated by the following expression with parallel operators.

$$PROGRAM1 = A \mid\mid B \mid\mid C \mid\mid D \mid\mid E \mid\mid F$$

Process PROGRAM2 runs a set of parallel processes with successive priority. This is illustrated by the following expression with pri-parallel operators.

$$PROGRAM2 = A \overleftarrow{\mid\mid} B \overleftarrow{\mid\mid} C \overleftarrow{\mid\mid} D \overleftarrow{\mid\mid} E \overleftarrow{\mid\mid} F$$

To illustrate the behavior of these two compositions, each process outputs two messages; one prefix message before communication and a suffix message after communication. The processes outputting on a channel are characterized as writers and the processes inputting from a channel are characterized as readers. More specific, the set of writer processes $W = \{ A, C, E \}$ and the set of reader processes $R = \{ B, D, F \}$ are defined as follows:

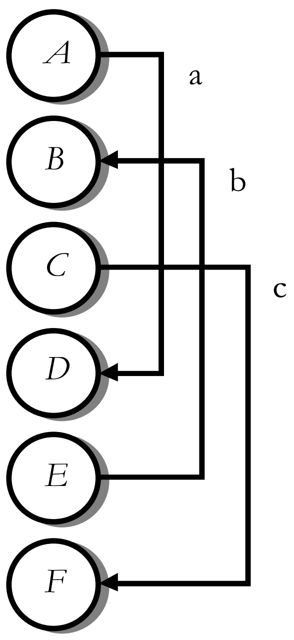

Figure 7 Network of processes

Each writer process $W_i \in W$ is defined as:

```
Wi = Wi′; (channeli ! any → Wi″) ; process behaviour
Wi′= print(Wi.name + “ ′”)       ; print process name and append ′ character (prefix message)
Wi″= print(Wi.name + “ ″ ”)      ; print process name and append ″ character (suffix message)
```
where W.name = { “A”, “C”, “E” } and channel = { a, c, b }.
W_i.name represents the name property of W_i.

Each reader process $R_i \in R$ is defined as,

```
Ri = Ri′; (channeli ? any → Ri″) ; process behaviour
Ri′ = print(Ri.name + “ ′”)      ; print process name and append ′ character (prefix message)
Ri″ = print(Ri.name + “ ″ ”)     ; print process name and append ″ character (suffix message)
```
where *R.name* = { “B”, “D”, “F” } and *channel* = { b, a, c }.
R_i.name represents the name property of R_i.

The composition process *PROGRAM1* in Java is given by listing 4.

```
program1 = new Parallel(new Process[]
{
  new Writer(a, "A"),
  new Reader(b, "B"),
  new Writer(c, "C"),
  new Reader(a, "D"),
  new Writer(b, "E"),
  new Reader(c, "F")
});
```
Listing 4 PROGRAM1 process

The process `program1` is the instance of the `Parallel` class. The composition process can be run by invoking its `run()` method.

```
program1.run();
```

The results of this process are given in the first column of table 1. The sequence of scheduling between processes is automatically determined at run-time. This non-preemptive scheduling behavior is purely based on channel communication.

The composition process `PROGRAM2` is almost similar, but process `program2` is the instance of the `PriParallel` class as listed in Listing 5.

```
program2 = new PriParallel(new Process[]
{
  new Writer(a, "A"),
  new Reader(b, "B"),
  new Writer(c, "C"),
  new Reader(a, "D"),
  new Writer(b, "E"),
  new Reader(c, "F")
});

program2.run();
```

Listing 5 PROGRAM2 process

The results of this process are given in the second column of table 1. The sequence of scheduling between processes is automatically determined at run-time. In this composition, the preemption of processes is performed according to decisions made by channel communication that is based on their priorities. A channel schedules the process with the highest priority first. Preemption of a lower priority process may also be performed at wake-up events of higher priority processes, i.e. higher processes that are woken-up after a sleep operation or after the occurrence of an interrupt. Scheduling on wake-up events is not illustrated in this paper, but is fully supported by `PriParallel` construct.

The sequence of processes in the process array of `program2` is determined by their priority. The output of `program2` is fixed. The sequence of processes in the process array of `program1` is undetermined and ambiguous in that the sequence may be arbitrary chosen. The output of `program1` is fixed for each sequence of processes.

A third process, *PROGRAM3*, illustrates another valid sequence of processes of *PROGRAM1*.

$$PROGRAM3 = F \;||\; E \;||\; D \;||\; C \;||\; B \;||\; A$$

The composition process of *PROGRAM3* is given by:

```
program3 = new Parallel(new Process[]
{
  new Reader(c, "F"),
  new Writer(b, "E"),
  new Reader(a, "D"),
  new Writer(c, "C"),
  new Reader(b, "B"),
  new Writer(a, "A")
});

program3.run();
```

Listing 6 PROGRAM3 process

Table 1 Results of test programs.

	Results of *PROGRAM1*	Results of *PROGRAM2*	Results of *PROGRAM3*
1	A’	A’	F’
2	B’	B’	E’
3	C’	C’	D’
4	D’	D’	C’
5	D”	A”	C”
6	E’	D”	B’
7	E”	E’	B”
8	F’	B”	A’
9	F”	E”	A”
10	A”	F’	F”
11	B”	C”	E”
12	C”	F”	D”

```
/*
 * Class name : ExampleParPriPar
 * Description:
 *
 * File Name  : ExampleParPriPar.java
 * Date       : 2/2/1999
 * Author(s)  : Gerald H. Hilderink
 */

// import classes from CTJ-library

import csp.lang.*;
import csp.lang.Process;
import csp.lang.System;

/**
 * Example process.
 **/

public class ExampleParPriPar
{
  public static void main(String[] args)
  {
    new ExampleParPriPar().run();
  }

  Process program1, program2, program3;

 /**
  * Create compositions of processes and channels.
  *
  * PROGRAM1 = A || B || C || D || E || F
  * PROGRAM2 = A <| B <| C <| D <| E <| F
  * PROGRAM3 = F || E || D || C || B || A
  **/

  public ExampleParPriPar()
  {
    Channel_of_Any a = new Channel_of_Any();
    Channel_of_Any b = new Channel_of_Any();
    Channel_of_Any c = new Channel_of_Any();

    //-- Parallel composition construct for PROGRAM1

    program1 = new Parallel(new Process[]
    {
      new Writer(a, "A"),
      new Reader(b, "B"),
      new Writer(c, "C"),
      new Reader(a, "D"),
      new Writer(b, "E"),
      new Reader(c, "F")
    });

    //-- PriParallel composition construct for PROGRAM2

    program2 = new PriParallel(new Process[]
    {
      new Writer(a, "A"),
      new Reader(b, "B"),
      new Writer(c, "C"),
      new Reader(a, "D"),
      new Writer(b, "E"),
      new Reader(c, "F")
    });

    //-- Parallel composition construct for PROGRAM3

    program3 = new Parallel(new Process[]

    {
      new Reader(c, "F"),
      new Writer(b, "E"),
      new Reader(a, "D"),
      new Writer(c, "C"),
      new Reader(b, "B"),
      new Writer(a, "A")
    });
  }

 /**
  * Process body.
  **/
```

```
  public void run()
  {
    System.out.println("PROGRAM1");

    program1.run();

    System.out.println("PROGRAM2");

    program2.run();

    System.out.println("PROGRAM3");

    program3.run();
  }
}

/**
  * Writer process: W = W';(channel!any -> W")
  **/

  class Writer implements Process
  {
    ChannelOutput_of_Any channel;   // local channel
    String name;                    // process name

    public Writer(ChannelOutput_of_Any chanout,
                  String processname)
    {
      channel = chanout;
      name    = processname;
    }

   /**
    * Process body.
    **/

    public void run()
    {
      try
      {
        System.out.println(name+"'");
        channel.write();
        System.out.println(name+"\"");
      }
      catch (csp.io.IOException e) { }
    }
  }

 /**
  * Reader process: R = R';(channel?any -> R")
  **/

  class Reader implements Process
  {
    ChannelInput_of_Any channel;    // local channel
    String name;                    // process name

    public Reader(ChannelInput_of_Any chanin,
                  String processname)
    {
      channel = chanin;
      name    = processname;
    }

   /**
    * Process body.
    **/

    public void run()
    {
      try
      {
        System.out.println(name+"'");
        channel.read();
        System.out.println(name+"\"");
      }
      catch (csp.io.IOException e) { }
    }
  }
}
```

Listing 7 Source code of the test programs

The output of `program3` is given in third column of table 1. In theory, both processes *PROGRAM1* and *PROGRAM3* are equal. Although the outputs of `program1` and `program3` are different, they are valid outputs. Every arbitrary sequence of the process array in a parallel construct results in a valid output.

The scheduler works on one Java thread. The core Java synchronization can block this thread and thus can block the entire scheduling. So, do not use the Java synchronization mechanism. You can synchronize with the Java threads via some special methods. The full code of program1, 2 and 3 is given in listing 7.

5. Embedded Process 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 generalise 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. As a consequence different concurrent programs can be scheduled in their 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:

The scheduler object can be included when needed. When more than one scheduler of the same type is needed only one code segment resides in memory.

Different types of schedulers can be used. The scheduling behavior can be nested such that logistic and real-time policies can be mixed.

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.

The open interface of this approach permits other design patterns for concurrent programming and scheduling policies.

With the monitors one can write robust programs according to certain design patterns [11]. 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. 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 [10]. 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 realisation of real time requirements as follows:

1. The non-deterministic behaviour of Java that is caused by cloning objects or object serialisation, 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 be reused efficiently and the process behaviour will be deterministic. However, this copying concept does not conflict with the cloning and serialisation concepts of Java, which can be used as well. However it does not require the need of garbage collection.
2. Special link drivers extend the scheduling behaviour 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 utilisation [9].
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 optimised 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.

7. 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 channel between processes rather than of processes 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 implemented CSP channels in Java satisfy our needs of a distributed Real-Time Java system, including the use of priorities according to the Rate Monotonic scheduling algorithm. External interrupts are scheduled at the prescribed priorities. External I/O should be triggered externally by means of a hardware clock and not by a software triggering.

The beauty of the system is that it contains clearly defined real-time principles such as real-time scheduling and priorities, programmed in an object oriented language. The drawback is that the present Java implementation is too slow to be called a real-time system. The C version that was derived using this Java version as a design pattern operates two orders of magnitude faster than the Java version. We are convinced however that the use of state of the art light weight threads could result in a speed-up of three orders of magnitude. Until this is accomplished by Sun, Java cannot seriously be considered to have real-time possibilities. In the mean time we use the C version of the system which allows us to port the code to a large number of processors.

References

[1] C.A.R. Hoare, “Communicating Sequential Processes”, Prentice Hall International Series in Computer Science, 1985. ISBN 0-13-153271-5 (0-13-153289-8 PBK).

[2] A.W. Roscoe, “The Theory and Practice of Concurrency”, Prentice Hall International Series in Computer Science, 1997. ISBN 0-13-674409-5.

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

[4] P. H. Welch, “Java Threads in the Light of CSP”, *Proceedings of the WoTUG-21 conference, 5-8 April 1998, Canterbury, UK. pp 259-284*

[5] Paul Austin, “Java Communicating Sequential Processes (JCSP) Library” URL: http://www.hensa. ac.uk/parallel/ languages/java/jcsp/.

[6] A.R. Hendriks, “Design of a Realtime and Embedded Scheduler in Java’, MSc thesis, report nr. 057R98, October 1998, Control Laboratory, University of Twente

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

[8] L. Sha, et al., “The Priority Inheritance Protocol: An Approach to Real-Time Synchronization”, Department of Computer Science, Carnegie-Mellon University Pittsburg PA, 1987.

[9] 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.

[10] 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

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

[12] A. E. Lawrence, “Extending CSP”, *Proceedings of the WoTUG-21 conference, 5-8 April 1998,Canterbury, UK. pp 111-131*