A Distributed Real-Time Java System Based on CSP

Gerald Hilderink, André Bakkers, and Jan Broenink
University of Twente, P.O.Box 217, 7500 AE Enschede
Control Laboratory, The Netherlands
G.H.Hilderink@el.utwente.nl

Abstract

CSP is a fundamental concept for developing software for distributed real-time systems. The CSP paradigm constitutes a natural addition to Object Orientation and offers higher-order multithreading constructs. The CSP 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 has been carefully examined and as a result it was reasoned that priority scheduling should be attached to the communicating channels rather than to the processes. In association with channels, a priority based parallel construct is developed for composing processes – hiding threads and priority indexing from the user. This approach simplifies the use of priorities for the object-oriented paradigm. 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.

1. Introduction

The concept of CSP – ‘Communicating Sequential Processes’ as introduced by Hoare [1] and more recently brought up to date by Roscoe [2] and Schneider [3] – constitutes a formalism that may be used to design distributed real-time Java programs. The CSP paradigm constitutes a natural addition to object-orientation. In fact the CSP channel concept is a natural way to use multithreading without being troubled even once with threads programming. The thread administration is completely handled by the CSP channel. In fact the CSP addition renders object-orientation to become truly concurrent without the additional burden of threads programming. The resulting code is not only easy to program and reason about, it is also safe to use because it is based on the rules of CSP, which guarantees the correct interaction between concurrent processes.

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 CSP channels replace the method calls 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 by using channels. The question one may ask is whether this channel concept holds ground for real-time aspects like scheduling and priority handling. This paper defines the notion of priority and scheduling and it shows 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.

Section 2 describes the fundamental programming concepts of the CSP paradigm. Section 3 described the channel concept for the Java programming language. The notion of priority as a real-time property, is discussed in section 4. Moreover, additional aspects of real-time programming are discussed in section 5. Conclusions are given in section 6.

2. The CSP Paradigm

CSP is a notation for describing concurrent systems whose component processes interact with each other by communication. In [4] is illustrated that CSP provides an excellent means of describing and reasoning about complex communication patterns, for the following reasons.

- It encapsulates the fundamental principles of communication in a simple and elegant manner.
- It is semantically defined in terms of a structured mathematical model, which may be used to deduce system properties rigorously.
- It is sufficiently expressive to enable reasoning about the pathological problems of deadlock and livelock.
- The principles of abstraction and refinement are central to the underlying theory.
- Robust software engineering tools exist for formal verification in CSP.

Certain mainstream programming languages, such as occam, Ada and some dialects of parallel C are derived from the CSP model. CSP-style communications libraries are available for other languages, such as Java.

The next sections describe the fundamental programming concepts of the CSP paradigm. In section 2.1 five CSP compositional constructs are discussed and in section 2.2 the CSP channel is highlighted.
2.1. Composite processes

In concurrent software, tasks are executed in parallel. A process describes a composition of tasks that belong to that process. A process is not necessarily a sequential task. A process can be composed of other simpler processes. The simplest process is a sequential task. Processes may run in parallel, in some sequence or by some choice. CSP specifies fundamental operators that describe the sequence of executing processes. These operators provide compositional constructs, such as: PAR for parallel, SEQ for sequential, or ALT for alternative. This is illustrated in example 1.

<table>
<thead>
<tr>
<th>SEQ</th>
<th>PAR</th>
<th>ALT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process1</td>
<td>Process1</td>
<td>Guard1: Process1</td>
</tr>
<tr>
<td>Process2</td>
<td>Process2</td>
<td>Guard2: Process2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Processn</td>
<td>Processn</td>
<td>Guardn: Processn</td>
</tr>
</tbody>
</table>

**Example 1. Fundamental compositional constructs.**

The SEQ construct determines that processes listed under this construct are executed in sequential order. The SEQ construct is completed after the execution of all processes listed under the SEQ construct. 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. The ALT construct consists of guards that in turn each guard a process. There are several types of guards: input guard, output guard, skip guard and timeout guard, which can either be conditional or unconditional. As soon as a guard becomes ready and when selected by the ALT, the guarded process is executed, completing the execution of the ALT. A guard is unconditional.

<table>
<thead>
<tr>
<th>PRI PAR</th>
<th>PRI ALT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process1</td>
<td>Guard1: Process1</td>
</tr>
<tr>
<td>Process2</td>
<td>Guard2: Process2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Processn</td>
<td>Guardn: Processn</td>
</tr>
</tbody>
</table>

**Example 2. Extended fundamental compositional constructs.**

CSP is limited to processes with equal priorities. However, CSP includes the PRIALT construct as a legal ALT with a prioritized selection algorithm (example 2). The guards of a PRIALT construct are given a selection priority, the first guard is the highest priority, the last the lowest. When two or more guards are ready then the one with the highest priority will be selected. As soon as a guard becomes ready and selected by the ALT, it the guarded process is executed, completing the execution of the PRIALT.

Lawrence [5] has extended the theory of CSP with notion of priority for processes and formally describes the PRIPAR construct (example 2). The PRIPAR construct determines that processes listed under this construct are executed concurrently but with different priority of execution. Each process executes at a separate priority, the first process is the highest priority, the last the lowest. Lower priority processes may only continue when all higher priority processes are unable to continue. The PRIPAR construct is completed after the execution of all processes listed under the PRIPAR construct.

A process describes the behavior of an object in terms of the events in which it may engage. An event is an atomic occurrence that happens in time. An event is not an instance of any class. Examples of events are: the successful termination of processes, occurrence of interrupts, timeouts, exceptions, the wakeup of sleeping processes, and message passing between processes. These events can be modeled in terms of communication, where processes interact with each other and their environment. The most fundamental object in CSP is therefore a communication event. These communication events act on objects that are known as channels.

2.2. What are CSP channels?

The basic idea of CSP as defined by Hoare [1] is that parallel or concurrent processes may work together by synchronizing on the respective inputs and outputs of these processes. The communication from process A to process B can only occur if process A is ready to send a message to process B and at the same time process B is ready to receive the message 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. The pseudo-code in example 3 illustrates two communicating processes via a channel.

```
CHANNEL chan
PAR
  ProcessA(chan)
  { chan ! x; ... }
  writing x to chan
  ProcessB(chan)
  { chan ? y; ... }
  reading y from chan
```

**Example 3. Two communicating processes.**

CSP exclusively uses channels to realize communication between processes. In fact, processes only communicate by means of channels. Channels control synchronization and scheduling of these processes. Channels are one-way, initially unbuffered, and fully synchronized. However, buffer processes may be added to make the communication asynchronous if one so desires.

An important property of the use of CSP channels is the fact that the resulting program scales well with the complexity of the system. That is, complex systems are just as easily implemented as the popular two-process examples, see next section.

---

1 This is in contradiction with the Java AWT event handling where events are instances of the Event class that are passed to objects on the occurrence of an event. It is recognized that Java AWT and Swing does not very well cooperate with multiple threads. Something went wrong.
3. CSP for Java

At present, two CSP-style packages for Java, JCSP [6] and CTJ [7], provide a CSP concurrency model for Java. Both packages are built on top of the Java thread model, whereas CTJ contains a special embedded scheduler for real-time properties. Both packages are developed with real-time and embedded systems in mind and are viable candidates for Real-Time Java. These packages show that the CSP concept is an excellent basis for concurrent programming in object-oriented programming languages such as Java, for the following reasons:

- The process- and channel-oriented paradigm is object-oriented by nature.
- Processes are orthogonal, compositional, anonymous to other processes, and distributive in that they can run anywhere.
- Channels are simple synchronization primitives that provide communication between concurrent or distributive processes.
- Priority is encapsulated within processes and is compositional in the same way processes do.
- Channels separate hardware/software concerns.

In the next sections we use our CTJ library (available at http://rt.el.utwente.nl/javapp) to illustrate these features by examples. Section 3.1 described the CSP channel as implemented in Java and section 3.2 describes the link driver concept for controlling hardware by channel communication.

3.1. 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 processes that are active objects. This is illustrated in figure 1. Processes may only read or write on channels. A communication event occurs when both processes are ready to communicate. Processes get blocked until communication is ready. Channels send entire objects by copying their contents from source object to the destination object. Sharing objects on a shared memory system is also possible by passing reference. Synchronization, scheduling, and the physical data transfer are encapsulated in the channel. The result is that the programmer is freed from complicated synchronization and scheduling constructs.

Figure 1. Data flow at design level: process oriented.

The channel model reduces the gap between concurrent design models, such as data-flow models, and the implementation. In data-flow diagrams, an arrow corresponds to a channel and a circle represents a process. Figure 1 represents a data-flow at design level, whereas figure 2 represents the implementation. In figure 2 the arrows corresponds to the flow of control (or invocation on objects) and the annotation arrow at the bottom of the figure corresponds to arrow in the data-flow diagram of figure 1. Process A invokes a write method on the channel object and process B invokes a read method on the channel object.

```
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 Channel_of_Integer class defines the channel object that only accepts Integer objects. It consists of the ChannelInput_of_Integer interface, specifying the read(object:Integer) method and the ChannelOutput_of_Integer interface, specifying the write(object:Integer) method. The main() method acts as a so called network builder, which typically declares channels and processes and executes the processes in parallel. Such network builder maps a data-flow design to implementation and reverse – one can draw a data-flow diagram from the implementation. A distributed version of the program contains a Main class for each processor. The Parallel construct contains only those processes for that processor and the channels are declared with knowledge about the external link that connects the systems. Important is that, except for the network builder processes, other processes stay unchanged. This approach eliminates anomalies and results in good maintainability and extensibility of the software. Moreover, code generation for concurrent software becomes straightforward.

In listings 2 and 3, process A (producer process) and process B (consumer process) are given. Process A produces 10,000 incrementing integer numbers starting from zero. Process B consumes these 10,000 numbers and prints them onto the screen.
public class ProcessA implements Process
{
    Channel Output_of_Integer channel;
    Integer object;

    public ProcessA(Channel Output_of_Integer out)
    {
        channel = out;
        object  = new Integer();
    }

    public void run()
    {
        while(object.value < 10000) {
            object.value++;
            channel.write(object);
        }
    }
}

Listing 2. Producer ProcessA.

public class ProcessB implements Process
{
    Channel Input_of_Integer channel;
    Integer object;

    public ProcessB(Channel Input_of_Integer in) {
        channel = in;
        object  = new Integer();
    }

    public void run()
    {
        while(object.value < 10000) {
            object.value++;
            System.out.println(object.value);
        }
    }
}

Listing 3. Consumer ProcessB.

The process classes define their input/output interfaces by their constructors. This allows us to assemble processes together via channels as a plug-and-play concept of components (or building blocks). 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. Naturally, one-to-one and many-to-one relations can be realized. A one-to-many (broadcasting) relation needs a separate design pattern, e.g. the delta building block that is explained in [8].

3.2. The Link Driver Concept

The channel concept in Java deals with all aspects of communication. The CTJ 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.

Hardware access is seen as low-level communication with a hardware process on the device. At a higher level, this hardware process is part of communication itself and therefore it is part of channel communication. The CTJ channel can contain a special driver that encapsulates the hardware dependent code for controlling the link. Channels between processes on one processor use a shared memory driver and channels between processes on different processors use a peripheral driver. There will be a clear separation between hardware dependent and hardware independent code as illustrated in figure 3.

Figure 3. Plug and Play framework for devices.

To avoid the development of special channels for each peripheral, a device driver framework has been developed. These device drivers, so-called link drivers, are hardware dependent objects that can be plugged into the channel. The read(…) and write(…) methods of the channel are delegated to the read(…) and write(…) methods of the link driver. The channel synchronizes the threads that are invoking its read(…) and write(…) methods, i.e. process A will wait until the buffer state of the link driver is not full and process B will wait until this state is not empty. This frees the link driver from doing synchronization, but the link driver may perform specific synchronization, that is provided by the link driver framework, as well.

Figure 4. Data transfer uni-processor systems.

Figure 4 shows communication between two processes on one processor, whereas figure 5 shows communication between two distributed systems, using the TCPIP protocol.

Figure 5. Data transfer for multi-processor systems.

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

```java
public class ProcessA implements Process
{
    Channel Output_of_Integer channel;
    Integer object;

    public ProcessA(Channel Output_of_Integer out)
    {
        channel = out;
        object  = new Integer();
    }

    public void run()
    {
        while(object.value < 10000) {
            object.value++;
            channel.write(object);
        }
    }
}

Listing 2. Producer ProcessA.

public class ProcessB implements Process
{
    Channel Input_of_Integer channel;
    Integer object;

    public ProcessB(Channel Input_of_Integer in) {
        channel = in;
        object  = new Integer();
    }

    public void run()
    {
        while(object.value < 10000) {
            object.value++;
            System.out.println(object.value);
        }
    }
}

Listing 3. Consumer ProcessB.

The process classes define their input/output interfaces by their constructors. This allows us to assemble processes together via channels as a plug-and-play concept of components (or building blocks). 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. Naturally, one-to-one and many-to-one relations can be realized. A one-to-many (broadcasting) relation needs a separate design pattern, e.g. the delta building block that is explained in [8].

3.2. The Link Driver Concept

The channel concept in Java deals with all aspects of communication. The CTJ 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.

Hardware access is seen as low-level communication with a hardware process on the device. At a higher level, this hardware process is part of communication itself and therefore it is part of channel communication. The CTJ channel can contain a special driver that encapsulates the hardware dependent code for controlling the link. Channels between processes on one processor use a shared memory driver and channels between processes on different processors use a peripheral driver. There will be a clear separation between hardware dependent and hardware independent code as illustrated in figure 3.

Figure 3. Plug and Play framework for devices.

To avoid the development of special channels for each peripheral, a device driver framework has been developed. These device drivers, so-called link drivers, are hardware dependent objects that can be plugged into the channel. The read(…) and write(…) methods of the channel are delegated to the read(…) and write(…) methods of the link driver. The channel synchronizes the threads that are invoking its read(…) and write(…) methods, i.e. process A will wait until the buffer state of the link driver is not full and process B will wait until this state is not empty. This frees the link driver from doing synchronization, but the link driver may perform specific synchronization, that is provided by the link driver framework, as well.

Figure 4. Data transfer uni-processor systems.

Figure 4 shows communication between two processes on one processor, whereas figure 5 shows communication between two distributed systems, using the TCPIP protocol.

Figure 5. Data transfer for multi-processor systems.

Declaring a channel with a link driver is illustrated by the following code:
Therefore, priority relations cannot be encapsulated within an object, because it is of a global object-oriented. The meaning of an index value cannot be with explicit indexing is that the resulting constructs are not issue and not a design issue. The most important problem indexing is considered to be a scheduler implementation i.e. equal to (=) or greater than (>) relations. Priority relations are global. An important problem with the threads of control. Such explicit indexing of priorities has a global character; the meaning of an index and the priority relations between indexes expresses the priority relations between processes as found in a data-flow diagram. Each priority relation is a dashed line labeled with a symbol {=,>,<} between two processes. A group of processes can also be seen as one composed process.

4. The notion of Priority

In our field, control systems are real-time systems that mostly have to operate within hard real-time constraints. Periodic and aperiodic tasks must be scheduled to meet specific deadlines. The run-time scheduling mechanism must be able to guarantee that the most critical deadlines are met even if this is at the expense of missing less important deadlines. The existence of hard real-time constraints introduces a new level of complexity into the design of programming languages. To deal with this extra dimension the notion of priority is widely used. 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. Problems such as the priority inversion problem must be dealt with. So actually it is the communication channel that is burdened with the task to resolve the differences in priority. Section 4.1 discusses the powerful expression of the PRIPAR for composing priorities as an object-oriented concept. The choice of scheduling behavior we used to implement the PRIPAR is motivated in section 4.2. Section 4.3 explains some of the implementation detail of the scheduler that is embedded in the application by the PRIPAR. The solution to the priority inversion problem for the channel concept is discussed in section 4.4.

4.1. Composing priorities

In many programming languages, as in Java, priority is an index assigned to a thread of control. The comparison between indexes expresses the priority relations between the threads of control. Such explicit indexing of priorities has a global character; the meaning of an index and the priority relations are global. An important problem with explicit indexing is that designer must determine the index values (absolute priorities) by global knowledge, whereas the designer is only interested in relative priority relation, i.e. equal to (=) or greater than (>) relations. Priority indexing is considered to be a scheduler implementation issue and not a design issue. The most important problem with explicit indexing is that the resulting constructs are not object-oriented. The meaning of an index value cannot be encapsulated within an object, because it is of a global nature. Therefore, priority relations cannot be encapsulated within objects. For example, an existing object that contains multiple threads with different priorities is used in a new project is likely to be re-indexed when used with other objects containing threads. This means that existing objects must be updated with new indexes. This anomaly can be avoided by relative indexing with priority offsets, which makes code unnecessary complex.

This section proposes the PRIPAR construct that was discussed in section 2. This construct acts like a PAR construct with the additional requirement that the sequence of the processes listed under the PAR construct also gives the descending order of the priority. Clearly there should be no limit to the number of processes under the PRIPAR construct and even nesting of PRIPAR’s is allowed as in example 4.

Example 4. A nested PRIPAR construct.

The priority relations between the processes of example 4 are depicted by the priority graph in figure 6. A priority graph shows the essential priority relations between processes as found in a data-flow diagram. Each priority relation is a dashed line labeled with a symbol {=,>,<} between two processes. A group of processes can also be seen as one composed process.

Figure 6. Priority graph.

The PRIPAR construct is based on implicit indexing and relative priorities that is superior to explicit indexing as describe above. Priority indexes are hidden from the user and the priority relations are relative to the processes within the PRIPAR construct. Priorities of processes can change at run-time by moving processes in the list of processes. Adding and removing processes is based on the process identity and not necessarily by their index.

4.2. 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) algorithm and Deadline Driven (DD) algorithms. The RM algorithm has been selected for the PRIPAR construct for a number of reasons. The RM algorithm,
which uses a fixed priority list, has in its unrestricted form the disadvantage that the CPU utilization 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 overhead costs makes it less attractive in a real-time environment. In real-time systems, there is a control-engineering-theoretical need to have measurements coincide with each other. Thus the ratio of the maximum \( m_i \) sampling time and the \( i \)th sampling time becomes \( 2^m \), which is an integer. The consequence of this is that the CPU utilization for a RM algorithm 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. The combination of the PAR construct, PRIPAR construct and channels allows priority scheduling to be either preemptive or non-preemptive. The moments of terminating processes in the PAR or PRIPAR and blocking on channel communication are events on which is being scheduled. As a result, the scheduling behavior of processes with RM can be traced, i.e. debugging processes. Also, analysis of the behavior of processes with static priorities (RM) may become a supplement for CSP in the future [5]. This makes the RM algorithm a logical choice for implementation [10]. Nevertheless, DD scheduling algorithms, such as early deadline first or stack resource protocol algorithms are interesting topics. We believe it is possible to construct these scheduling algorithms in a compositional form similar as the PRIPAR, e.g. EDF PAR or SRP PAR.

4.3. Embedded Scheduling

The PRIPAR construct uses a special scheduling concept. The scheduler consists of a set of classes that is not part of the Java Virtual Machine but is embedded in the application. The scheduler contains a dispatcher that is responsible for determining which process will run next. Special is that the dispatcher is a process that can run under another dispatcher. This way we achieve a nestable scheduler. Internally, processes are kept up by queues and move from queue to queue. The dispatcher manages the ready queue and the running queue. Each channel manages its own waiting queue. A timer link driver manages the timing queue for letting processes sleep or for performing time-outs.

Each PRIPAR constructor assigns a new dispatcher to its parent dispatcher. The PAR does not create any dispatcher except when it is the first process of the program, as in listing 1. At least one dispatcher is needed when only PAR constructs are used. Each embedded dispatcher may perform its own special scheduling algorithm. We have implemented a RM dispatcher for the PRIPAR, but we expect that also DD dispatchers are possible (section 4.2). The PAR construct provides a first-come, first-served (FCFS) algorithm. In combination with the PRIPAR with a higher-priority process, it turns into a round-robin (RR) algorithm. For example:

```
PRI PAR
  TimeSlicer(timeQuantum) {
    while(true) { sleep(timeQuantum) }
  }
PAR
  Process1
  Process2
```

Example 5. A time slicing construct.

Every time the Ti meSlicer process \( (P_a) \) wakes-up after the specified time quantum, it preempts the PAR. The preempted process, Process1 \( (P_1) \) or Process2 \( (P_2) \), will be placed on the ready queue and after \( P_a \) sleeps again the next ready process, \( P_2 \) or \( P_1 \), will run. The sleep statement performs a wake-up event after termination – an internal channel communicates with the hard real-time clock. This way time slicing is optional and this can be implemented in CSP manner. It is trivial to see that time slicing can be nested. Note that we hardly ever use time slicing in real-time systems. The priority graph of figure 7 illustrates the priority relations. We assume that \( P_1 \) and \( P_2 \) form an existing group of processes.

![Figure 7. Priority graph of a time slicing system.](image)

Our Java implementation of the PRIPAR, which is defined by the PriParallel class, may have a maximum of 7 processes, because every PRIPAR is being decoded in one byte with one bit per priority. The eighth bit is used for idle tasks, skip tasks, or pre-emptive garbage collection. The latter has not been implemented yet, but real-time garbage collection has been taking into account in its specification. In a nested composition of PRIPARs, an infinite number of priorities can be achieved.

4.4. Priority inversion problem

Should a high-priority process \( (P_i) \) be blocked on a channel, waiting for communication with a process of lower priority \( (P_j) \), it may have to wait a longer time than seems reasonable, because a third process \( (P_k) \) of middling priority might be hogging the CPU (figure 8). In order to do justice to overall system performance, it would be reasonable to elevate the scheduling priority of \( P_i \) to the level of the blocked \( P_j \). This results in the well-known priority inversion problem [11], where the priority graph is combined within a data-flow diagram.
Processes deal with channels, not other processes. If the channel is empty (i.e. the process trying to communicate gets blocked), there is no way to find out who has the lock and raise its priority when required? It’s different with monitors. If a thread is blocked trying to get a lock on a monitor, the monitor can find out who has the lock and raise its priority if necessary. After releasing the monitor the priority is restored to its original priority.

Priority inversion is a silly problem – one that comes from bad design in the first place. From the point of view of the higher-priority process $P_h$, the last thing it wants is the priority of another process $P_l$ to be raised to it so that this process competes with it – the higher-priority process has real-time duties to service! Priorities are set for a reason. A hard real-time design rule is: don't communicate with a lower-priority process unless you don't have any real-time guarantees to deliver. Feel free to communicate with a lower-priority process (and maybe get blocked) if you currently have no real-time service commitments.

The solution to solve priority inversion is different to priority inheritance, which follows a general design pattern: give the high-priority real-time servicing process an equal-priority buddy process. When a high-priority servicing process needs to communicate with a lower-priority process, get its buddy process to do it. The buddy process is listening out for the servicing process – so the servicing process won’t be blocked communicating with its buddy. The buddy then gets blocked (maybe) communicating with the lower-priority process but no matter – the high-priority process has real-time duties to service! The buddy then gets blocked (maybe) communicating with the lower-priority process but no matter – the high-priority process has real-time duties to service!

Priority inversion is a silly problem – one that comes from bad design in the first place. From the point of view of the higher-priority process $P_h$, the last thing it wants is the priority of another process $P_l$ to be raised to it so that this process competes with it – the higher-priority process has real-time duties to service! Priorities are set for a reason. A hard real-time design rule is: don't communicate with a lower-priority process unless you don't have any real-time guarantees to deliver. Feel free to communicate with a lower-priority process (and maybe get blocked) if you currently have no real-time service commitments.

The solution to solve priority inversion is different to priority inheritance, which follows a general design pattern: give the high-priority real-time servicing process an equal-priority buddy process. When a high-priority servicing process needs to communicate with a lower-priority process, get its buddy process to do it. The buddy process is listening out for the servicing process – so the servicing process won’t be blocked communicating with its buddy. The buddy then gets blocked (maybe) communicating with the lower-priority process but no matter – the high-priority process is still alive and servicing. The servicing process needs to remember not to communicate with its buddy until its buddy communicates back after dealing with the lower-priority process. If this is necessary, some more buddies are needed.

The above design pattern needs no priority raising. The high-priority buddy can get stuck communicating with a low-priority process. Fine, the buddy hasn’t got anything else to do! The buddy process needs to be equal-priority with the servicing process so that the buddy will succeed as soon as the low-priority process is ready to communicate with it so that it gets the attention of the servicing process when that’s been done.

A simple design pattern that eliminates priority inversion is a buffer process as the buddy process as depicted in figure 9a. The higher-priority process writes to the buffer and can immediate continue without being blocked. After servicing it may wait so that lower-priority processes continue to consume the messages. Aside, sub-sampling or over-sampling buffers can be used in place for the connection between hard real-time and soft real-time processes.

In CTJ, buffer processes can be implemented as buffered channel, i.e. a channel with a passive buffered link driver implemented (figure 9b). This approach saving context switches, need no extra channel, and avoids anomalies in design and implementation. The latter can be shown in that figure 9b is similar to figure 8a and that the difference is an instance of a buffer link driver passed to a channel.

Note, in CSP channels are unbuffered, i.e. rendezvous. With buffered channels one must add some corrections (i.e. adding a buffer process and an extra channel) to the formal CSP description. However, a deadlock-free program with rendezvous channels will be deadlock-free with buffered channels. The reverse may not be true [2].

5. Real-time aspects

Timing is one the most important 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 realization of real-time requirements as follows:

1. The non-deterministic behavior of Java that is caused by cloning objects and excessive garbage can be avoided when using channels. CTJ 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 behavior will be deterministic. It may only require the need of garbage collection on termination of the process.  

2. Special link drivers 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 belief this type of scheduler may be used to the full 100% CPU utilization [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 optimized for processes down to Java byte code.  

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 C and C++ version that is derived from the present Java implementation is ultra fast compared to the Java version. The C version runs on a small Texas Instruments TMS320F240 DSP controller. The C or C++ version can be a viable supplement to realize Real-Time Java.  

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 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 software using 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. 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.  

It is clear that the CSP channel concept provides a simple component-based development (CBD) methodology that holds a great promise [12].  

7. References  


