Communicating Java Threads

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

Abstract. The incorporation of multithreading in Java may be considered a significant part of the Java language, because it provides rudimentary facilities for concurrent programming. However, we belief that the use of channels is a fundamental concept for concurrent programming. The channel approach as described in this paper is a realization of a systematic design method for concurrent programming in Java based on the CSP paradigm. CSP requires the availability of a *Channel* class and the addition of composition constructs for sequential, parallel and alternative processes. The Channel class and the constructs have been implemented in Java in compliance with the definitions in CSP. As a result, implementing communication between processes is facilitated, enabling the programmer to avoid deadlock more easily, and freeing the programmer from synchronization and scheduling constructs. The use of the Channel class and the additional constructs is illustrated in a simple application.

1 Introduction

The programming language Java has multithreading capabilities for concurrent programming. To provide synchronization between asynchronously running threads, the Java language and runtime system uses *monitors* which are described in (Hoare, 1974). Clearly, within Java the concept of “*Communicating Sequential Processes*”, CSP (Hoare, 1978) has not implemented. In CSP channels and composition constructs are described to provide well-behaved communication between processes (i.e. threads). Instead, Java provides general concepts for implementing communication constructs that are as flexible as possible. This kind of freedom allows various kinds of communication facilities to be used. We will discuss these various communication facilities that are available and will show that the use of channels may result in a robust and structured approach for Communicating Java Threads (CJT).

We have developed a Channel class and classes for the sequential, parallel and alternative communication constructs. These classes not only result in better structured programs, but they are also founded on the theoretical basis of CSP (Hoare, 1985). As we speak of a channel (in lower-case) we refer to an instance of its Channel class (with a capital) which describes its definition.

Modern computers use highly integrated microprocessors, which are typically based on the *Von Neumann Model of Computation*. This model is characterized by the fact that instructions are executed successively. The advantage of writing sequential programs is that the code can be optimized for full speed execution. As a result, many programming languages inherit this single flow of control that forces sequential programming.

Developments in software architectures, such as Structured Design Methods (Ward and Mellor, 1985;Yourdon, 1989) and Object-Orientation (Rumbaugh et al., 1991), have shifted their focus to software architectures that emphasize the way in which human beings think about software, away from the manner in which the software is executed. Object oriented designs result in system descriptions in which every object has some task that is independent of what other objects do. In object oriented implementations it is customary to isolate the aspects of its task inside an object. One thing in this process is rather unnatural: when analysis and design models are taken into consideration, the notion of parallelism is thrown away. What looked like a parallel solution is *flattened* into a single flow of control. Object oriented programming languages describe this in terms of sequential dispatching of messages to objects. On the other hand, the real world is a place where many things happen concurrently. Following the principle of real world modeling, this should also show up in the resulting software. Besides, this activity of flattening a concurrent model into a sequential application is far from trivial and is the major cause of programming errors.

Java is a sequential programming language that features Object-Orientation and multithreading. However, multiple threads run in parallel processes and are created in a sequential fashion. This paper addresses the danger of deadlock that is caused by sequentially ordered communication between threads. In chapter 2 some aspects of concurrent programming are emphasized as we think are important. Some common communication facilities provided by Java are briefly described and compared to the channel facility in chapter 3. In chapter 4 the channel and composition constructs as a formal concept are discussed. In chapter 5 the channel implementation and composition constructs according to the object-oriented paradigm are described. A simple benchmark program that illustrates the use of channels is discussed in chapter 6.

2 Concurrent programming

A concurrent program is a single program that consists of several processes running in parallel. A *process* is a sequential program in execution. A concurrent program has control over its processes.

The human cognitive aspect of composing processes is very natural in developing programs. Processes are highly independent, because they are bounded to their task, and they provide fundaments of the structure of the program. In the field of object-orientation, processes can be considered as active objects that have their own life.

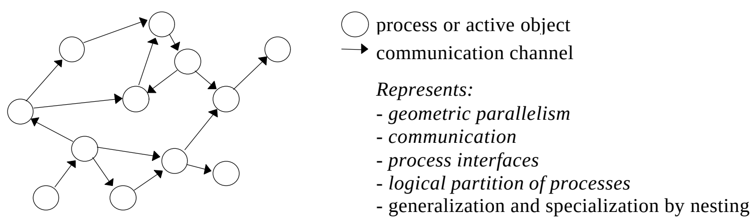


Figure 2.1: Data-flow diagram (directed graph)

The analysis phase of the software development determines – among other things – the tasks to be carried out by the program. In the design phase these tasks are mapped onto graphical models such as data flow diagrams, activity diagrams, state transition diagrams, flow charts etc. The data flow diagram is an important graphical model in the design phase. A data flow diagram is a directed graph, see for example figure 2.1, which defines the processes – the degree of parallelism – and the structure of the program. Each circle represents a process or a flow of control, whereas the arrows denote the data flow or communication between the processes. The graphical representation of a data flow diagram is a powerful model in the design phase, which represents processes at different levels of nesting, i.e. specialization and generalization.

It is the data flow model that is hard to flatten into a sequential implementation and as a result these types of models are often omitted in design tools. Modern developments, such as support of multithreading by programming languages, make concurrent programming more easy. The concept of multithreading makes the data flow model useful again – provides parallelism down to the code – and makes flattening superfluous.

2.1 Multithreading versus processes

The concept of multithreading within a sequential programming language, such as Java, allows us to create processes running in parallel. Processes and threads are conceptually related. A *process* is the actual program executed by one or more threads. The process maintains its program structure and state, i.e. objects and variables. A *thread* is a single flow of control that can execute its instructions independently. A thread gets scheduled on a CPU and comprises the contents of registers – the program counter, the general registers, the stack pointer, and the stack.

In (Lewis and Berg, 1996) the following example illustrates an essential point of threads:

> “Let's consider a human analogy: a bank. A bank with one person working in it (traditional process) has lots of "bank stuff" such as desks and chairs, a vault, and teller stations (process tables and variables). There are lots of services that a bank provides: checking accounts, loans, savings accounts, etc. (the functions). With one person to do all the work, that person would have to know how to do everything, and could do so, but it might take a bit of extra time to switch among the various tasks. With two or more people (threads), they would share all the same "bank stuff," but they could specialize on their different functions. And if they all came in and worked on the same day, lots of customers could get serviced quickly”.

This application is a traditional program or a single process run by multiple threads. Every “bank stuff” object is shared among several employees and must be synchronized to prevent simultaneous use, i.e. the “bank stuff” must be thread-safe. This synchronization is provided by a protocol that decreases the efficiency, which is reason of complexity, and causes intertwined code that is hard to maintain. This bottleneck can be reduced by supplying separate “bank stuff” for each employee, that is, creating more processes. Only one employee then uses each "bank stuff", i.e. the objects are not shared and there is no synchronization needed for every item. Only communication between the employees should be synchronized which will be less frequent.

In (Lewis and Berg, 1996) threads can make traditional programs concurrent by adding multithreading whereas we think programs should be constructed in terms of processes. Processes are active objects having their own life and may have multiple threads that run underlying and encapsulated passive objects. Traditional programs that are made multithreaded are still flattened into sequential programs. Real concurrent programs, consisting of processes, are scaleable and portable on multiprocessor systems with or without shared memory.

2.2 Communicating processes

Many authors assume that multithreading lead to higher performance and better-structured programs (Lewis and Berg, 1996). However, we have learned that multithreading doe not always guarantee well-behaved and/or better-structured programs. On the contrary, communication between threads (or processes) may often lead to complicated code, which makes programs more complex, more error-prone and less structured than necessary. We think that the use of *channels* and *composition constructs*, which are based on channels, contribute to well-behaved and ‘easy’ structured programs (i.e. to comprehend, verify, and formalize) that keeps the data flow model upright. Conspicuously, this approach does not conflict with other strategies or object-orientation in general. On the contrary, channels are objects that pass objects between objects. The latter are often active objects. With this channel concept the programmer can reason about the structure and the behavior of the program as if observing a data flow diagram.

3 Possible means of communication

Communication between processes can be established by means of shared memory or by *interprocess-communication* (IPC) facilities as elaborately described in (Silberschatz and Galvin, 1994). Java provides communication through shared-memory, message passing, input/output streams, and remote links. Channels are unknown in Java and are not discussed in (Silberschatz and Galvin, 1994), however, they are almost of the same age as those mentioned above. The main reasons that channels were omitted are that channels do not work in sequential environments (i.e. sequential languages and non-multitasking operating systems) and that the costs of context switching (i.e. CPU-switching between threads) were considered too expensive. However, the channel concept carried on in the field of parallel computing.

This section discusses the communication facilities provided by Java and discusses the channel facility, for as we think this is the best means for communication between processes. There are other communication facilities for *distributed* programs (e.g. Parallel Virtual Machine (PVM), Remote Method Invocation (RMI), and Common Object Request Broker (CORBA) which are out of the scope of this paper, but these facilities fit well in the channel paradigm.

3.1 Shared memory

In single-threaded programs data can efficiently be passed by shared memory. Globally declared variables or structures – shared objects – function as an asynchronous buffer pool between communicating threads. This asynchronous behavior may lead to incorrect results when multiple threads are trying to update the same data in an uncontrolled manner, e.g. data may not be overwritten by the writer until it is read by the reader. This *data inconsistency* can be avoided when the data updates are synchronized by means of semaphores, such as an atomic *test-and-set* of a shared boolean variable, which provides a protocol between the writers and readers (Dijkstra, 1968). The programmer should create a *critical region* around the shared object, which ensures that it can only be accessed by one thread of control at a time. Such a critical region can be created by using a *monitor* construct (Hoare, 1974) that is a high-level synchronization construct with powerful *wait* and *signal* operations (Silberschatz and Galvin, 1994). In Java the `synchronized-wait-notify` construct provides such a high-level monitor construct (Arnold and Gosling, 1996).

3.2 Message passing

In message-based programming languages there is no need to resort to shared variables. Messages are dispatched by a sender process to a receiver process which processes the message. In the object-orientation paradigm a message consists of a destination, a method name, and arguments (i.e. the data). Objects only communicate via message passing, that is, by invoking accessible methods on other objects. Shared methods that update variables should be *thread-safe* in that only one thread can invoke the method at one time. Java methods can be made thread-safe by using the `synchronized-wait-notify` construct for methods as for shared memory (Arnold and Gosling, 1996).

3.3 Input/output streams

Multithreading operating systems provide input/output (I/O) streams for communication between programs or to system resources, such as the file-system, standard input, standard output, and standard error. There are different I/O streams, such as data streams, file-streams, piped streams, etc. Streams are ordered sequences of data that have a source (input stream), or destination (output streams) and contain a cyclic buffer of memory. Streams are asynchronous, but are synchronized when a full or empty condition is reached; a sender process is blocked when the buffer is full and a receiver process is blocked when the buffer is empty. Streams are system resources – standardized by the ANSI normalization – and are often implemented for universal use. Streams are often restricted to an array of bytes and polling is needed to catch conditions such as end-of-file or errors. Java also provides I/O streams.

3.4 Remote links

Remote links are methods provided by the operating system, which can set-up a communication link between distributed objects. In contrast to I/O streams, remote links can send messages to other programs and can be seen as an extension of message passing over a longer distance between computer systems. Remote links -- also called remote procedure calls (RPC) -- provide an abstract procedure-call mechanism for use between systems with network connections. In contrast with the IPC facilities, the RPC communication facility provides a layer build on top of the operating system. There are already many solutions in the form of Local (or Wide) Area Networks or by object linking and embedding which we will not discuss here. Java provides network supports and Remote Method Invocation (RMI).

3.5 Channels

The channel connects processes and takes care of the data transfer. The channel is one-way and self-synchronized. Communication in both ways needs two channels. Initially, the channel is unbuffered and provides a rendezvous synchronization mechanism; in other words, a message can only be sent if both the writer and reader are ready. If one of the processes is ready and the other is not, then the first gets blocked until the second is ready. In other words, processes get scheduled on communication. In the object-oriented paradigm the channel is an intermediate object that encapsulates synchronization, scheduling and the data transfer. As a result the programmer is freed from synchronization, scheduling, and the physical data transfer. Our implementation of the channel allows for sharing among multiple readers and writers, and can also pass different data types as objects. Furthermore, the channel can be used in three different composition constructs, i.e. the sequential composition, the parallel composition, and the alternative composition (section 4.2).

3.6 Review

There are a few reasons why channels are not included in sequential programming languages. In the early years, operating systems where designed and build to schedule sequential programs or processes. The operating system was often the only shared resource. Scheduling was in those days very time consuming and as a result buffering communication is chosen over rendezvous communication (buffering reduces the number of context switching). For these reasons, we find I/O streams instead of channels. Nowadays, I/O streams are flexible, standardized and are typical operating system resources for many purposes, such as standard input, standard output, disk access, pipes, filters, etc. Another important reason is that the rendezvous behavior does not work in sequential programs. Channel input and output on the same channel by the same process and in a sequential composition, results immediately in deadlock whereas I/O streams may work. The performances of modern processors are high and the cost of scheduling is almost negligible. The technology of Object-Orientation and multithreading brings concurrency within the program itself. These factors make the use of channels within sequential programming languages very well possible.

The shared-memory and message passing concepts need explicit synchronization implemented by the programmer. The I/O streams and remote links are system resources that are restricted to sending and receiving bytes in a quasi-synchronous manner; I/O streams and remote links can only be used in the presence of an operating system that is supporting them. The channel is not a system resource and operates close to the scheduler.

For real-time systems one likes to avoid the operating system, because of space and performance. The channel can be optimized which makes scheduling very fast in a non-preemptive fashion that provides an optimal CPU utilization (Liu and Layland, 1973). By using channels, the scheduler could be build within the program itself and by which the program schedules itself in an non-preemptive fashion. The non-preemptive paradigm can also be mixed with preemptive scheduling for dealing with fair scheduling (e.g. priority scheduling).

Because the channel initially is unbuffered, the program will schedule more frequently than, for instance, a buffered I/O stream. In our channel we have created a *plug-in* feature in which the channel can be buffered for optimization purposes (e.g. reducing the number of context switching). The same feature can also be used for communication between processes on other microprocessors or systems by plugging in external links (section 5.1).

The use of channels and the composition constructs that are based on channels are further discussed in chapter 4 and the Java implementation will be discussed in chapter 5.

4 Channels and composition constructs

4.1 Channel formalism

The semantics of the rendezvous channel are straightforwardly defined and have their theoretical foundation in CSP. This theoretical foundation makes it possible to use automated tools for validating designs. Aside from being mathematically pleasing, the channel construct is easy to reason about, and therefore it will also be a beneficial construct to think of when designing software, making it a good object to identify during the object oriented design phase.

The channel has an input and an output method that are respectively written as $c?v$ and $c!v$ in the CSP language, where c is the channel name and v is the value of the message. In appendix A.1 some basic notations of the CSP language are described. If one process engages in $c?v$ and another process engages in $c!v$ then communication will be performed and value v will be passed by channel c.

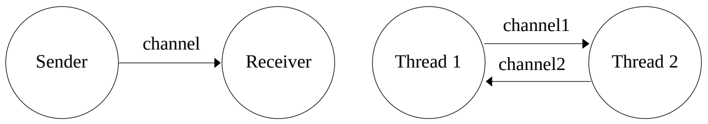


Figure 4.1: (a) one-way communication and (b) two-way communication

The channel provides one-way communication and two-way communication needs two channels. The only responsibility left with the programmer is to avoid *deadlock* by ensuring that the second thread becomes ready sometimes. Deadlock is the state in which two or more concurrent threads can no longer proceed due to communication interdependency. In appendix A.2, communicating processes are discussed in more detail.

Figure 4.1b shows a two-way or cyclic communication. Such a scheme can be prone to deadlock when performing input and output on channels in a sequential fashion.

4.2 Composition construct formalism

Appendix A.3 describes three compositions i.e. the (i) sequential, (ii) parallel, and (iii) alternative composition. A thread is a sequential composition, which executes statements in successive order. The `if-then-else` statement provides alternative compositions in which choices are made according to expressions.

The programmer should be careful when choosing the order of channel input and output in a sequential composition. For the convenience we take the CSP notation for channel input (?) and channel output (!).
A correct sequence of reading and writing for figure 4.1b is given by

```
Thread 1:   do sequential           Thread 2:   do sequential
            {                                   {
              channel1 ! x                        channel1 ? a
              channel2 ? y                        channel2 ! b
            }                                   }
```

Thread 1 writes the value x and thread 2 reads the value into a through channel1. In the next step, thread 2 writes the value b and thread 1 reads the value in y through channel2. If we choose a wrong sequence then deadlock may occur.
Consider for example,

```
Thread 1:   do sequential                  Thread 2:   do sequential
            {                                          {
               channel1 ! x  ← deadlock                   channel2 ! b  ← deadlock
               channel2 ? y                               channel1 ? a
            }                                          }
```

Both threads are blocked, because both write actions wait for communication, which will never occur. If the threads could read or write in parallel then we would have no deadlock and the order of execution is not important:

```
Thread 1:   do sequential                  Thread 2:   do parallel
            {                                          {
               channel1 ! x                               channel2 ! b
               channel2 ? y                               channel1 ? a
            }                                          }
```

A parallel construct at one side in this example is sufficient to prevent deadlock. However, the order of communication is indefinite and therefore a parallel construct at both sides is more obvious.

```
Thread 1:   do parallel                    Thread 2:   do parallel
            {                                          {
               channel1 ! x                               channel2 ! b
               channel2 ? y                               channel1 ? a
            }                                          }
```

Here, we have illustrated a simple example of two communicating threads, which can be scaled for more complex networks of channels and threads. In a network with more than one cycle, the parallel construct will be indispensable (Welch, 1987). In appendix A.2 we formally illustrate this example by means of CSP and we proof the appearance of deadlock. Appendix A.3 describes the behavior of the three composition constructs and will be illustrated in section 5.2.

5 Java channel and composition constructs

An important strategy of developing software is *separation of concerns* (Hürch and Lopes, 1994). Separation of concerns follows the well-established principle in software engineering to hide complexity by abstraction. Programming intertwined code is hard and complex since all concerns have to be dealt with at the same time and at the same level. Intertwined code is hard to understand, maintain and modify because the concerns are strongly coupled. The discipline of separating concerns is used throughout this work. Section 5.1 discusses the identity of classes on which the Java channel is build. Section 5.2 discusses the use of the channel in association with its composition constructs.

5.1 The Java channel

The interface of the channel defines a channel input and output operation that are visible to the user. The Java channel consists of more than one class and to ensure a simple interface we have build the Java channel as a synchronized adapter (Lea, 1996) according to the Adapter, Composite, and Proxy design patterns as described in (Gamma et al., 1995).

The Java channel is defined at the conceptual level by four abstractions of concerns:

Channel. The channel encapsulates synchronization, scheduling, and the data transfer into one object. The channel input and output will be delegated to the *link driver* that is part of the channel.

Link driver. The actual data transfer between processes can be performed internally (i.e. through shared memory) or externally (i.e. through shared communication lines). The concerns of the link driver are to perform the physical data transfer, to localize and to encapsulate the hardware dependent code from hardware independent code. In other words, link drivers are device drivers for communication purpose that are plugged into the channel.

Clonable protocol object. Clonable protocol objects encapsulate the information sent by channels. These objects have knowledge – a protocol – about which variable may or may not be cloned; i.e. copies the contents into a specified object.

Composition. Processes can invoke the read (channel input) and write (channel output) methods on the channel in a sequential, parallel or alternative fashion. The behavior of the channel input and output depends on the composition construct they use and on the type of synchronization provided by the link driver.

These concerns are reflected in the class relation diagram of figure 5.1 in OMT notation. The Channel class is related to zero or many composition members, zero or more clonable protocol objects, and to a link driver that is part of the channel. The channel can exist without a composition member or without a clonable protocol object, but contains a link driver. For reading and writing on the channel a composition member and a clonable protocol object must be specified at each side.

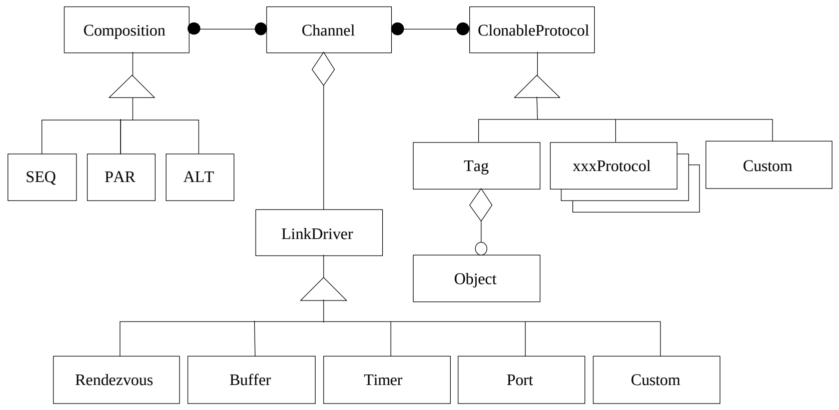


Figure 5.1: Class relation diagram

The SEQ, PAR and ALT classes, respectively the sequential, parallel, and alternative classes, inherit the Composition class. The ClonableProtocol class extends some standard protocol classes, such as the xxxProtocols (intProtocol, byteProtocol, charProtocol, shortProtocol, longProtocol, floatProtocol, doubleProtocol, and booleanProtocol) and the Tag. The ClonableProtocol class makes sure that the channel can clone the object otherwise the object will not be accepted. The Rendezvous, Buffer, Timer, and Port classes are standard link driver subclasses of the LinkDriver class. The LinkDriver class adds a `state` variable that must be updated by the link driver of which the channel behavior depends on. The super-classes ClonableProtocol and LinkDriver provide *generic* functionality that can extend the functionality of the channel easily. The Composition class is reserved for future features and it completes the diagram. The programmer can build his own clonable protocol object or link drivers without having knowledge of the implementation of the channel or the composition constructs.

5.1.1 Channel basics

The Java channel is a multiple readers/writers object, which will clone the protocol object according to the behavior defined by the link driver. If no link driver is specified then the rendezvous link driver will be the default link driver.

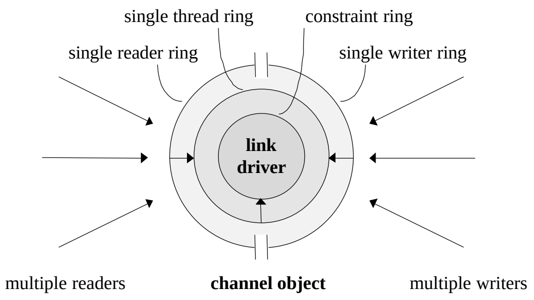


Figure 5.2: Channel monitor

Figure 5.2 shows a channel object of which the inner critical region, that is the link driver, is guarded by three rings of synchronization (monitors). The single reader and writer rings allow passing a single reader and a single writer. The single thread ring allows passing a single reader or a single writer. The constraint ring provides the behavior or the (de)scheduling of threads according to the state of the `state` variable updated by the link driver. The `state` variable can hold one of the three constants `FULL`, `EMPTY`, or `NOTEMPTYFULL` (read not empty and not full) which represents the actual state of the buffer inside the link driver; physically the link driver is buffered by a temporary variable or by an array of variables.

Here is a simple version of the Channel class defining a `read()` and a `write()` method. The Channel class provides the rings of synchronization and delegates the actual data transfer to the `read()` and `write()` methods of the link driver. The `linkdriver.state` variable is part of the constraint ring which provides the desired behavior specified by the link driver (rendezvous by default).

```
class Channel
{
  LinkDriver linkdriver = new Rendezvous();                 // default link driver

  public boolean read(ClonableProtocol object)
  throws Exception
  {
    boolean result = false;
    synchronized (reader_monitor)                           // single reader's ring
    {
      synchronized (this)                                   // single thread ring
      {
        if (linkdriver.state == LinkDriver.EMPTY) wait();   // constraint ring
        result = linkdriver.read(object);                   // delegate read to link driver
        notify();                                           // notify a waiting writer
      }
      return result;
    }
  }

  public boolean write(ClonableProtocol object)
  throws Exception
  {
    boolean result = false;
    synchronized (writer_monitor)                           // single writer's ring
    {
      synchronized (this)                                  // single thread ring
      {
        result = linkdriver.write(object);                 // delegate write to link driver
        notify();                                          // notify a waiting reader
        if (linkdriver.state == LinkDriver.FULL) wait();    // constraint ring
      }
      return result;
    }
  }
  …
}
```

Besides the variety of link drivers we can distinguish between four channel types. That is, the channel can be implemented with no single reader and writer ring, one reader ring, one writer ring, or both reader and writer rings which depend on the number of readers and writers sharing the channel. A design pattern such as a *factory* method could be applied which creates one of the four channels in conformance with the number of readers and writers (Gamma et al., 1995).

The link driver is a simple class without synchronization constructs. In the next listing, we show the rendezvous link driver that provides internal communication between processes.

```
public class Rendezvous extends LinkDriver
{
  private ClonableProtocol object;

  public boolean read(ClonableProtocol object)
  {
    this.object.clone(object);
    this.state = EMPTY;
    return true;
  }

  public boolean write(ClonableProtocol object)
  {
    this.object  = object;
    this.state   = FULL;
    return true;
  }
}
```

The variable `object` is the temporary buffer variable. The `state` variable is inherited by the `LinkDriver` class and is updated conform the buffer condition. The `clone(..)` method is a method invoked on the clonable protocol `object`. An example of the `intProtocol` class is,

```
public class intProtocol extends ClonableProtocol
{
  public int value;

  public void clone(ClonableProtocol object)
  {
    ((intProtocol)object).value = this.value;
  }
}
```

The complexity of the link driver depends on the media on which data must be passed. The link driver acts as a device driver for communication purpose. With this approach it is more easy to adjust the channel for external communication links such as RS-232, USB, PCI, VME, CORBA, RMI, and many more. In other words, other technologies concerning communication can be used through link drivers. The channel functions as a structural approach in concurrent programming.

5.2 The Java Communication constructs

In the previous section, the `read()` and `write()` methods of the channel are discussed. As mentioned, the behavior of these methods depends on the state of the link driver. This section discusses an extension of this behavior in which these methods can be used in different compositions of execution as discussed in section 4.2. Implementations of these compositions are respectively the SEQ, PAR, and ALT constructs. These abbreviations are borrowed from *occam* (INMOS, 1988). The `SEQ`, `PAR`, and `ALT` classes, shown in figure 5.1, provide methods for constructing the SEQ, PAR, and ALT compositions constructs in Java.
The `Channel` class also provides three other pairs of `read()` and `write()` methods for each composition construct. These methods invoke the `read()` and `write()` as described in the previous section but under different circumstances. Without discussing the entire implementation of these `read()` and `write()` methods, we will illustrate the use of these methods in the next sections of this chapter.

The methods provided by the Channel class are:

```
boolean channel.read(<construct member>, <clonable object>);
boolean channel.write(<construct member>, <clonable object>);
```

where *`channel`* is the name of a channel object, `<construct member>` is a composition object of `SEQ`, `PAR`, or `ALT`, and `<clonable object>` is a clonable protocol object. The `boolean` is the return type of the methods.

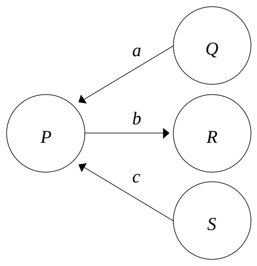


Figure 5.3: Example

The following examples of this section illustrate the three composition constructs for process *P* in figure 5.3. Process *P* has three shared channels *a*, *b*, and *c* that are declared as:

```
Channel channel_a = new Channel();
Channel channel_b = new Channel();
Channel channel_c = new Channel();
```

when `channel_a`, `channel_b`, and `channel_c` are channel objects.
The clonable protocol objects used to receive or to send are,

```
intProtocol object_a = new intProtocol();
intProtocol object_b = new intProtocol();
intProtocol object_c = new intProtocol();
```

5.2.1 The sequential communication construct

The sequential construct (SEQ) provides sequential channel input and output; the read and write actions will behave sequentially within the method.

```
SEQ seq = new SEQ();

channel_a.read(seq, object_a);
channel_b.write(seq, object_b);
```

```
channel_c.read(seq, object_c);
```

The `seq` object provides knowledge of doing sequential reading and writing.

5.2.2 The parallel communication construct

The parallel construct (PAR) provides parallel channel input and output. This construct is based on Dijkstra's *parbegin* (Dijkstra, 1968) which specifies concurrent execution of its constituent sequential commands (processes).

```
PAR par = new PAR(3);

channel_a.read(par, object_a);
channel_b.write(par, object_b);
channel_c.read(par, object_c);
par.join();
```

The `par` object provides knowledge of doing reading and writing in parallel. The `read()` and `write()` methods in a PAR construct start-up a hidden processes and immediately continue. The `par` object provides a `join()` method that will block until the parallel `read()` and `write()` methods are done. The number `3` in the PAR constructor specifies three hidden processes that carry out the `read()` and `write()` methods in parallel.

5.2.3 The alternative communication construct

The alternative construct (ALT) makes choices, which are based on the state of channel input or output. The `read()` and `write()` methods in the ALT-construct return true if communication is succeeded, else they return false. The read operation is called the input guard and the write operation is called the output guard of the ALT-construct.

```
ALT alt = new ALT();

if (a.read(alt, object_a))
{
  ... THEN program part
}
else
  if (b.write(alt, object_b))
  {
    ... THEN program part
  }
  else
    if (c.read(alt, object_c))
    {
      ... THEN program part
    }
```

This example is a typical priority ALT-construct. There are other alternative constructs, such as the fair-ALT and select-method, but basically they are based on the `if-then-else` principle. One should be careful using output guards, because an input guard that meets an output guard is likely to lead to disagreement (Jones, 1987). Although at any time it may appear that a distant peer is prepared to communicate, it may already have committed itself to not doing so. For any channel, there only may be *one* process ALTing on it, although there may be many processes sharing the other end.

6 Application: simple Communication Timer benchmark

In this section we describe a simple concurrent program in Java to illustrate the use of channels and constructs as described before. The program, shown as a data flow diagram in figure 6.1, is a simple Communication Timer test program, named ComsTime, which is an up-counter that starts from zero. This Java program is a benchmark for measuring process (i.e. thread) context-switch times.

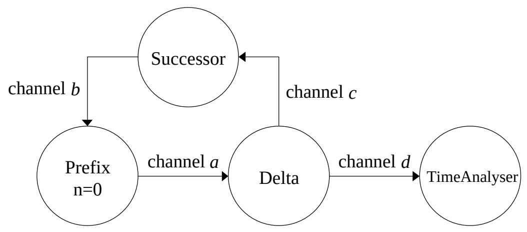


Figure 6.1: Simple concurrent program named "ComsTime"

This benchmark consists of four processes (Prefix(0), Delta, Successor, and TimeAnalyser) connected by channels. Each process or *building block* is an active object processed by a single thread of control whereas a channel is a passive object with no life of its own – its methods being called as part of threads from active objects. A building block is a thoroughly tested object which is influenced by its direct neighbors and is automatically scheduled when its inputs are available or its outputs are processed (Welch, 1988).

The Prefix(0) process kicks off by outputting 0 on its output channel and then going into a infinite *input-and-forward* cycle. The Delta process just cycles by waiting for input and then forwarding two copies (in parallel) to each of its output channels. The Successor process cycles through waiting for input, incrementing the number that arrives and forwarding the result to its output channel. Finally, the TimeAnalyser process simply consumes the arriving numbers (which will be the sequence of natural numbers) and times how fast they arrive.

The Prefix(0) and Successor processes synchronize with other threads twice per cycle, the Delta process three times and the TimeAnalyser process once per cycle. This makes 8 synchronizations (or context switches) per number that are consumed by the TimeAnalyser process. Dividing the cycle time reported by TimeAnalyser by 8 gives us the basic overhead for thread synchronization. The time spent by the processes themselves doing other things being negligible.

6.1 ComsTime implementation

Benchmark ComsTime, shown in listing 6.1, is a sequential program that declares channels, variables, and the building block objects. The building blocks are defined as threads and they are started automatically by its constructor. The `try-catch` body will intercept possible exceptions.

```
/**
 * ComsTime.java
 * Communication Timer Test Program
 **/

public class ComsTime
{
  public static void main(String args[])
  throws java.io.IOException
  {
    try
    {
      /* Test command line         */

      if (args.length != 2)
      {
         System.out.println("\nUsage: java ComsTime inner_loop_count outer_loop_count\n");
         System.exit(0);
      }
      /* Declarations: channels    */

      Channel channel_a = new Channel("channel a");
      Channel channel_b = new Channel("channel b");
      Channel channel_c = new Channel("channel c");
      Channel channel_d = new Channel("channel d");

      /* Declarations: variables   */

      int innerloop = Integer.parseInt(args[0]);
      int outerloop = Integer.parseInt(args[1]);

      /* Declarations: processes (the threads start automatically) */

      Prefix       prefix     = new Prefix(0, channel_b, channel_a);
      Delta        delta      = new Delta(channel_a, channel_c, channel_d);
      Successor    successor  = new Successor(channel_c, channel_b);
      TimeAnalyser timeAnalys = new TimeAnalyser(channel_d, innerloop, outerloop);

    } catch (Exception e)
      { System.out.println("Exception (main): "+e); }
  }
}
```

Listing 6.1: ComsTime

Programming in terms of processes (or building blocks) has some major advantages, which we will illustrate by means of listing 6.1.

There is a straightforward transition between the data flow model (figure 6.1) and the implementation (listing 6.1). The structure defined by the model specifies the structure of the implementation of the program and visa versa. One can easily draw a data flow diagram from the declarations of channel and building block objects. The transition could be automated by means of a generator that generates Java code from the data flow model.

The main program is a kind of network configurer, which builds a network of building blocks and channels. A network configurer is characterized in that it does not contain non-terminating loops, but starts-up the actual program. Afterwards the network configurer terminates successively. The Prefix building block also functions as a network configurer, which starts the Identity building block and connects it to the network. The idea of a network configurer has two advantages: (1) the programmer can browse easily through the program starting with the main program, and (2) the program can be partitioned for multiprocessor platforms.

The implementations of the network configurer as well as the building blocks are described in an orthogonal manner. A modification of the structure of the data flow model means adding or removing building block and channel declarations to or from the network configurer without influencing other parts of the program. The structure of the program can be very well maintained and extended by concatenation of other network configures. The network configurer and the building blocks are also very well reusable.

6.2 Building blocks implementation

The building block classes are subclasses of the Thread class. Statement `this.start()` in its constructor body automatically starts the object as a process (or active object) The `run(){}` body will be invoked by a the newly launched thread of control.

The parameters of the constructors are initially variables or shared channels; the shared channel parameters provide an channel interface for the object. In other words, building blocks have a method interface and a channel interface. These objects can be used through its methods or through its channel inputs and outputs.

6.2.1 Identity

The building block identity is a procedure with an input and an output parameter, both in the form of channels. An (intProtocol) object entering via the input channel is passed unchanged to the output channel. The procedure identity is identical to an active buffer with length one.

```
public class Identity extends Thread
{
  private ChannelInput  in;
  private ChannelOutput out;
  private SEQ           seq = new SEQ();
  private intProtocol   object = new intProtocol();

  public Identity(ChannelInput in, ChannelOutput out)
  {
    this.in  = in;
    this.out = out;
    this.start();
  }

  public void run()
  {
    try
    {
      while(true)
      {
        chanin.read(seq, object);
        chanout.write(seq, object);
      }
    } catch (Exception e)
      { System.out.println("Exception (Identity): "+e); }
  }
}
```

6.2.2 Prefix

We can use building blocks to make new building blocks. For example, the building block, called prefix, starts-up the building block identity (section 6.2.1). The procedure first sends out a specified number *n* (here *n* = 0) and then prefix behaves as identity. The building block prefix is an example of an configurer that starts building block identity after it initializes the output channel by sending a zero first.

```
public class Prefix extends Thread
{
  private ChannelInput  in;
  private ChannelOutput out;
  private SEQ           seq = new SEQ();
  private intProtocol   object = new intProtocol();

  public Prefix(int n, ChannelInput in, ChannelOutput out)
  {
    this.in  = in;
    this.out = out;
    object.value = n;
    this.start();
  }

  public void run()
  {
    try
    {
      out.write(seq, object);

      Identity id = new Identity(in, out);

    } catch (Exception e)
      { System.out.println("Exception (Prefix): "+e); }
  }
}
```

6.2.3 Delta

Another useful building block is the delta. Delta has one input and two output channels. Each incoming (intProtocol) object is passed unchanged to each of the two output channels in parallel. In this case the input object will be cloned to the two output objects. This building block is used whenever a process has to send a value to two other (parallel) processes. Delta or a similar process is required in such cases when two parallel processes may not read from the same channel and is as such the answer to broadcasting.

```
public class Delta extends Thread
{
  private ChannelInput  in;
  private ChannelOutput out1, out2;
  private SEQ           seq    = new SEQ();
  private PAR           par    = new PAR(2);
  private intProtocol   object = new intProtocol();

  public Delta(ChannelInput in, ChannelOutput out1, ChannelOutput out2)
  {
    this.in   = in;
    this.out1 = out1;
    this.out2 = out2;
    this.start();
  }

  public void run()
  {
    try
    {
      while(true)
      {
        in.read(seq, object);

        out1.write(par, object);
        out2.write(par, object);
        par.join();
      }
    } catch (Exception e )
      { System.out.println ("Exception (Delta): "+e); }
  }
}
```

6.2.4 Successor

The building block successor is largely the same as the building block identity. There is only one difference: the value of an (intProtocol) object that enters through the input channel is increased by 1 before being passed along to the output channel.

```
public class Successor extends Thread
{
  private ChannelInput  in;
  private ChannelOutput out;
  private SEQ           seq = new SEQ();
  private intProtocol   object = new intProtocol();

  public Successor(ChannelInput in, ChannelOutput out)
  {
    this.in  = in;
    this.out = out;
    this.start();
  }

  public void run()
  {
    try
    {
      while(true)
      {
        in.read(seq, object);
        ((intProtocol)object).value++;
        out.write(seq, object);
      }
    } catch (Exception e )
      { System.out.println ("Exception (Successor): "+e); }
  }
}
```

6.2.5 TimeAnalyser

Building block TimeAnalyser reads a burst of objects from the input channel and prints the delay time on screen. The `System.exit(0)` statement ends the whole program when TimeAnalyser is finished.

```
public class TimeAnalyser extends Thread
{
  private ChannelInput  chanin;
  private int           count1, count2;
  private intProtocol   object = new intProtocol();
  private SEQ           seq = new SEQ();

  public TimeAnalyser(ChannelInput ci, int cnt1, int cnt2)
  {
    chanin = ci;
    count1 = cnt1;
    count2 = cnt2;
    this.start();
  }

  public void run()
  {
    int i, j;
    long tstart, tstop, tdiff, tsum;

    System.out.println("Time Analyser started !\n");
    System.out.println("------------------------------");

    try
    {
      tsum = 0;
      for (j=0;j<count2;j++)
      {
        tstart = System.currentTimeMillis();
        for (i=0;i<count1;i++)
          chanin.read(seq, object);
        tstop = System.currentTimeMillis();
        tdiff = tstop - tstart;
        tsum  = tsum + tdiff;
      }
      System.out.println("Average of one cycle = "+((float)tsum)/(float)(count1*count2)
+" ms");
    } catch (Exception e)
      { System.out.println("Exception (TimeAnalyser): "+e); }
    System.exit(0);
  }
}
```

7 Conclusions

The `csp` classes form a complete package that contains the necessary ingredients for concurrent programming with channels in Java.

The channel and compositions concepts are derived from CSP and are developed according to the object oriented paradigm. There is a clear pattern of concerns by means of object oriented techniques using inheritance, delegation, genericity, and polymorphism. The channel itself is created according to the Adapter, Composite, and Proxy design patterns as described in Gamma et al. (1995).

The most important property of using channels and composition constructs is that the programmer is freed from synchronization and scheduling constructs. This facilitates programming of concurrent programs for the programmer.

Applications based on channels get scheduled on communication on a non-preemptive fashion. Non-preemptive scheduling is more efficient than preemptive scheduling and the channel can be optimized for better performances.

The channel and compositions concepts do not interfere or clash with other language concepts in Java. Furthermore, the channel and composition primitives could be integrated at byte-code level in Java as we see in occam.

Implementing software with channels eliminates the process of flattening the data flow model into one sequential program. Important is that there is no large gap, just a transformation, between the data flow model and the code. As a result, parallelism is maintained down to the code.

References

Arnold K., Gosling J.A., (1996), *The Java Programming Language*, Addison-Wesley, Massachusetts, USA

Davies J. and Schneider S., (1995), *Real-time CSP*, UK

Dijkstra E.W., (1968), *Co-operating sequential processes*, Programming Languages, Academic Press, New York, pp. 43-112

Gamma E. et al., (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesly, Massachusetts, U.S.A.

Hatley D.J., Pirbhai I.A., (1988), *Strategies for Real-Time System Specification*, Dorset House, New York, USA

Hoare C.A.R, (1985), *Communicating Sequential Processes*, Prentice-Hall, London, UK

Hoare C.A.R., (1974), *Monitors: An Operating System Structuring Concept*, Communication of the ACM, Vol. 17, No. 10, pp. 549-557

Hoare C.A.R., (1978), *Communicating Sequential Processes*, Communications of the ACM, Vol. 21, No. 8, pp. 666-677

Hürsch W.L. and Lopes C.V., (1994*), Separation of Concerns: Towards a New Paradigm of Software Engineering*, Boston, USA

INMOS, (1988), *occam 2 Reference Manual*, Prentice-Hall, London, UK

Jones G., (1987), *On Guards*, Parallel Programming of Transputer Based Machines, IOS

Lea D., (1996), *Concurrent Programming in Java*, Addison-Wesley, Massachusetts, USA

Lewis Bill and Berg J. Daniel, (1996), *Threads Primer: A guide to multithreaded Programming*, Sun Microsystems Press, Mountain View, USA

Liu C.L., Layland J.W., (1973), *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, the Association for Computing Machinery, No. 20(1), pp. 46-61

Martin J.M.R., (1996), *The Design and Construction of Deadlock-Free Concurrent Systems*, Ph.D. Thesis, University of Buckingham, UK

Rumbauch J., Blaha M., Premerlani W., Eddy F., Lorensen F., (1991), *Object-Oriented Modeling and Design*, Prentice-Hall, NJ., USA

Silberschatz Abraham and Galvin Peter B., (1994), *Operating Systems Concepts*, Addison-Wesley Publishing, Fourth Edition

Ward P.T. and Mellor S.J., (1985), *Structured Development Techniques for Real-Time Systems*, 3 vols, Prentice-Hall, NJ, USA

Welch P.H., (1987), Emulating Digital Logic using Transputer Networks, *Lecture notes in computer science*, Vol. 1, Springer-Verlag

Welch P.H., (1988), *An occam approach to transputer engineering*, Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications, Pasadena, CA, USA

Yourdon E.N., (1989), *Modern Structured Analysis*, Prentice-Hall, Englewood Cliffs, N.J., UK

Appendix A CSP channels and compositions

The process algebra *Communicating Sequential processes* (CSP) by (Hoare, 1978) is an algebraic language in which the behaviour of communicating processes can be described and can be verified by means of events and operators.

In this appendix we prove by the formal method of CSP the profit of the parallel composition for communicating sequential processes. Section A.1 describes some important mathematical notations of the CSP language. A rule is also given which can be used to prove if the process satisfies the desired behaviour. In section A.2, two processes are described by the notations of section A.1; the first process is an example of deadlock and the second process eliminates deadlock. The rule is used to prove the occurrence of deadlock for both examples. Section A.3 describes the behaviour of three common composition, i.e. (1) the sequential composition, (2) the parallel composition, and (3) the alternative composition, in CSP.

A.1 Process algebra

In CSP a process describes the behaviour pattern of an object in terms of events, operators, and other processes. To include events in our process description, we use the prefix operator:

if a is an event and P is a process, then

$(a \rightarrow P)$

is a process which may engage in a before behaving as P. An event can also be a communication event, which is represented by the pair

$c.v$

where c is the name of the channel on which the event takes place, and v is the value of the message that is passed. The input and output are respectively the read action $c?v$ and write action $c!v$ which are performed when both processes are ready and communication over channel c is performed. We define an input process

$(c?v \rightarrow P(v))$

as a process that behaves as process $P(v)$ after the value v is read by channel c. The value v is of type T denoted as $(c?(v : T) \rightarrow P(v))$. We define an output process

$(c!v \rightarrow P)$

as a process that behaves as process P after the value v is sent by channel c. This tightly defined communication protocol is called *rendezvous* communication. The set of messages communicable on channel c is defined as

$type(c) = \{v \mid c.v \in \alpha P\}$

The alphabet αP is the set of events in process P. The type of value v is specified by the output process $(c!v \rightarrow P) = (c.v \rightarrow P)$ and must be accepted by the input process $(c?x \rightarrow P(x)) = \square_{v:type(c)}(c.v \rightarrow P(v))$, where symbol $\square$ denotes choices. The output process defines one type and the input process accepts one or more types.

The processes *P* and *Q* can be observed in various compositions:

$(P \,;\, Q)$	**sequential composition**; behaves as *P* until this component is successfully terminated and then behaves as *Q*.
$(P \parallel Q)$	**parallel composition**; a composition in which *P* is able to engage in any event from the set αP, and *Q* is able to engage in any event from the set αQ. The two processes must cooperate to perform any event which is common to both alphabets. This parallel composition can also be described by $P \,|[\, A \,]|\, Q$ with $A = \alpha P \mid \alpha Q$ being the shared set of events.
$(P \,|||\, Q)$	**parallel composition (interleaving)**; the two processes *P* en *Q* evolve independently without cooperating upon every occurrence of any event of *P* and *Q*. This parallel composition is equal to $P \,|[\, A \,]|\, Q$ with set *A* is empty. If one process cannot engage in the action, then it must be the other one; but if both processes could have engaged in the same action, the choice between them is non-deterministic.
$(P \,\square\, Q)$	**alternative composition**; behaves as *P* if the first action of *P* can engage, else behaves as *Q* if the first action of *Q* can engage. If both actions can engage then the choice between them will be non-deterministic (the environment can control which of *P* or *Q* will be selected).
$(P \sqcap Q)$	**alternative composition (non-deterministic)**; the choice between *P* and *Q* is based on the arbitrary selection, without the knowledge or control of the external environment.
STOP	The process which never engages in any event. This describes the behaviour of a broken object; a deadlocked object.
SKIP	A process, which does nothing but terminates successfully.

An important rule in CSP, which we will use later in this section, is

$$
\begin{array}{llcl}
\text{Let} & P & = & \square_{x:X} \quad x \rightarrow P_x \\
 & Q & = & \square_{y:Y} \quad y \rightarrow Q_y \\
\text{Then} & P \,|[\, A \mid B \,]|\, Q & = & \square_{z:Z} \quad z \rightarrow (P_z' \,|[\, A \mid B \,]|\, Q_z') \\
 & \text{where } P_z' & = & P_z \text{ if } z \in X; P \text{ otherwise} \\
 & \text{and } Q_z' & = & Q_z \text{ if } z \in Y; Q \text{ otherwise} \\
 & \text{and } Z & = & (X \cap Y) \cup (X - B) \cup (Y - A) \\
 & \text{assuming} & & X \subseteq A \text{ and } Y \subseteq B
\end{array}
$$

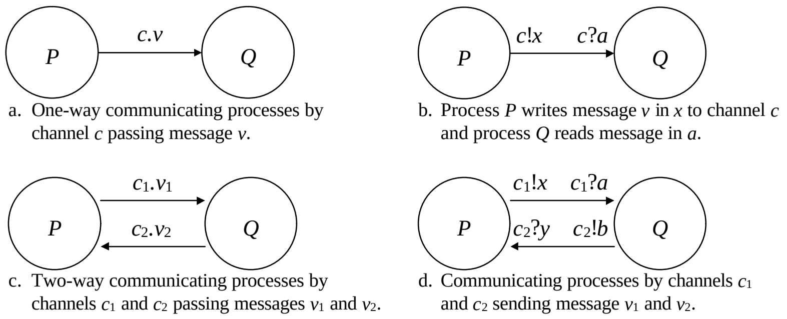
□

Rule 1.1: General choice operator rule

This rule provides an algorithm to evaluate the behavior of two parallel processes (P || Q). The shorthand notation $P = \square_{x:X} \; x \rightarrow P_x$ is a series of choices depending on the events *x* in set *X*. For instance, $P = \square_{x:\{a,b,c\}} \; x \rightarrow P_x$ is equal to $P = (a \rightarrow P_a) \,\square\, (b \rightarrow P_b) \,\square\, (c \rightarrow P_c)$. The set *Z* is defined as the collection of elements in *X* and *Y* and without double elements. The notation $(X - B)$ is the set $\{x \mid x \in X, x \notin B)$.

A.2 Communicating processes

The processes *P* and *Q* communicate through shared channels. The read and write actions on a channel are considered shared events that are performed when they are both ready. A channel provides one-way communication between two processes. For two-way communication two channels should be used. A channel that is used only for output by a process will be called an output channel of that process and one used only for input will be called an input channel.



a. One-way communicating processes by channel *c* passing message *v*.

b. Process *P* writes message *v* in *x* to channel *c* and process *Q* reads message in *a*.

c. Two-way communicating processes by channels c_1 and c_2 passing messages v_1 and v_2.

d. Communicating processes by channels c_1 and c_2 sending message v_1 and v_2.

Figure A.1: Communicating processes

Figure A.1 shows one-way and two-way communication between processes. Figure A.1a. shows a parallel composition where *P* en *Q* share the same channel *c*. Communication will occur on channel *c* on each occasion that *P* outputs a message and *Q* inputs that message, simultaneously. An outputting process specifies an unique value for the message, whereas the inputting process is prepared to accept any communicable value. Thus the event that will actually occur is the communication *c.v* , where *v* is the value of the message specified by the outputting process. Process *P* executes action *c*!*a* and process *Q* executes action *c*?*a* as shown in figure A.1b, in which the variables *x* and *a* hold the message *v*. Figures A.1c. and A.1d. show this for two-way communication between *P* and *Q*.

In case of two-way communication or in the presence of a cycle, we must avoid deadlock at any time. The next section will discuss the appearance of deadlock caused by a typical sequential order of communication between processes.

A.2.1 Sequential composition is deadlock sensitive

A thread is a sequential flow of control. The processes *P* and *Q* can be considered threads of control that invoke read and write actions on both channels each in a sequential orders. If this order is not correctly chosen by the programmer or by the flow of control then deadlock may be the result. Both processes will then wait on communication infinitely. The Processes *P* and *Q* of figure A.1c are once again shown in figure A.2. For convenience, the read i.e. $c_i?v$ and the write i.e. $c_i!v$ events are abbreviated to one event c_i with the same channel name. For instance, if the process engages in c_1 then process *P* engages in $c_1!v$ and process *Q* engages in $c_1?v$, that is, they are both ready to communicate.

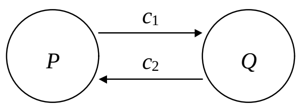


Figure A.2: Cyclic communicating processes

This system process of figure A.2 can be described as

P:
```
do sequential
{
  c1 ! x
  c2 ? y
}
```

Q:
```
do sequential
{
  c2 ! b
  c1 ? a
}
```

or in the CSP language,

$$SYSTEM \quad = \quad (P \parallel Q) = (\, P \,|[\, \alpha P \mid \alpha Q \,]|\, Q \,) \qquad \text{with } \alpha P = \{c_1,c_2\} \text{ and } \alpha Q = \{c_1,c_2\}$$

in which the processes *P* and *Q* are described as

$$P \quad = \quad (c_1 \rightarrow c_2 \rightarrow P')$$
$$Q \quad = \quad (c_2 \rightarrow c_1 \rightarrow Q')$$

The processes *P*' and *Q*' are suffix processes of respectively *P* and *Q* after the composition. We simply see that $c_1 \rightarrow c_2$ and $c_2 \rightarrow c_1$ only can be performed when $c_1 = c_2$. This is never true and therefore the processes *P* and *Q* are mutual exclusive or deadlocked.

In the next derivation we prove, on the basis of the rule 1.1 in section A.1, that this order of communication immediately leads to deadlock.

Theorem: Sequential communication in an erroneous order will lead to deadlock.

PROOF: We give a system that is equal to *STOP*.

Let

$$P \quad = \quad (c_1 \rightarrow c_2 \rightarrow P') \quad = \quad (c_1 \rightarrow P_{c1}) \quad \text{with} \quad P_{c1} = (c_2 \rightarrow P')$$
$$Q \quad = \quad (c_2 \rightarrow c_1 \rightarrow Q') \quad = \quad (c_2 \rightarrow Q_{c2}) \quad \text{with} \quad Q_{c2} = (c_1 \rightarrow Q')$$

then

$$SYSTEM \quad = \quad (\,(c_1 \rightarrow c_2 \rightarrow P') \,|[\, \{c_1,c_2\} \mid \{c_1,c_2\} \,]|\, (c_2 \rightarrow c_1 \rightarrow Q')\,)$$

following rule 1.1

$$X = \{c_1\},\ Y = \{c_2\},\ \text{and } Z = \{\}$$

with

$$P \quad = \quad \square_{x:\{c1\}} \quad x \rightarrow P_x \quad = \quad c_1 \rightarrow P_{c1}$$
$$Q \quad = \quad \square_{y:\{c2\}} \quad y \rightarrow Q_y \quad = \quad c_2 \rightarrow Q_{c2}$$

and

$$P \,|[\, \alpha P \mid \alpha Q \,]|\, Q \quad = \quad \square_{z:\{\}} \quad z \rightarrow (P_z' \,|[\, A \mid B \,]|\, Q_z') \quad = \quad STOP$$

or

$$SYSTEM \quad = \quad STOP$$

□

Process *SYSTEM* will never engage in any event of *P* and *Q* and will immediately deadlock. This proof can be extended to more than two communicating processes.

A.2.2 Parallel compositions eliminate deadlock

The read and write actions of process *P* and the read and write actions of process *Q* are mutual independent. The read and write action can be executed in arbitrary order because the order is actually undefined. As mentioned before, in a sequential composition, such as in a sequential process, the order of the read and write actions should be carefully chosen. A more secure approach is using a parallel composition which eliminates deadlock.

Consider example

```
P:   do sequential        Q:   do parallel
     {                         {
       c1 ! x                    c2 ! b
       c2 ? y                    c1 ? a
     }                         }
```

In the following proof we show that this system process always performs all actions. Again, this proof can be extended to more than two communicating processes.

Theorem: A parallel composition will prevent an erroneous order of communication and eliminates potential deadlock.

PROOF: We will show that a parallel composition always satisfies the behaviour of the system.

Let

$$P = (c_1 \to c_2 \to P') = (c_1 \to P_{c1}) \quad \text{with} \quad P_{c1} = (c_2 \to P')$$

$$Q = (c_2 \to c_1 \to Q') \square (c_1 \to c_2 \to Q')$$

$$= (c_2 \to Q_{c2}) \square (c_1 \to Q_{c1}) \quad \text{with } Q_{c1} = (c_1 \to Q') \text{ and } Q_{c2} = (c_2 \to Q')$$

then

$$SYSTEM = (c_1 \to c_2 \to P') \,|[\, \{c_1,c_2\} \mid \{c_1,c_2\} \,]|\, ((c_2 \to c_1 \to Q') \square (c_1 \to c_2 \to Q'))$$

following rule 1.1

$$X = \{c_1\},\ Y = \{c_1,c_2\}, \text{ and } Z = \{c_1\}$$

with

$$P = \square_{x:\{c1\}} \quad x \to P_x = c_1 \to P_{c1}$$

$$Q = \square_{y:\{c1,c2\}} \quad y \to Q_y = (c_1 \to Q_{c1}) \square (c_2 \to Q_{c2})$$

and

$$P \,|[\, \alpha P \mid \alpha Q \,]|\, Q = \square_{z:\{c1\}} \quad z \to (P_z' \,|[\, A \mid B \,]|\, Q_z') = c_1 \to (P_{c1} \,|[\, A \mid B \,]|\, Q_{c1})$$

The next behavior is

$$X = \{c_2\},\ Y = \{c_1,c_2\}, \text{ and } Z = \{c_2\}$$

with

$$P_z' = \square_{x:\{c2\}} \quad x \to P_x = c_2 \to P_{c2}$$

$$Q_z' = \square_{y:\{c1,c2\}} \quad y \to Q_y = (c_1 \to Q') \square (c_2 \to Q')$$

and

$$P \,|[\, \alpha P_z \mid \alpha Q_z \,]|\, Q = \square_{z:\{c2\}} \quad z \to (P_z'' \,|[\, A \mid B \,]|\, Q_z'') = c_2 \to (P' \,|[\, A \mid B \,]|\, Q')$$

or

$$SYSTEM = c_1 \to c_2 \to (P' \,|[\, A \mid B \,]|\, Q')$$

$\square$

Process *SYSTEM* can engage in both events c_1 and c_2 of *P* and *Q* and therefore this process satisfies the behavior of the system process.

In case where read and/or write actions are mutual independent, these actions should be enclosed in a parallel composition. The programmer should not have to worry about the order of execution. The previous example should be written as,

```
P:   do parallel          Q:   do parallel
     {                         {
       c1 ! x                    c2 ! b
       c2 ? y                    c1 ? a
     }                         }
```

The system process with parallel compositions at both processes is defined as

$$SYSTEM \quad = \quad (c_1 \rightarrow c_2 \rightarrow (P' \,|[\, A \mid B \,]|\, Q')) \;\square\; (c_2 \rightarrow c_1 \rightarrow (P' \,|[\, A \mid B \,]|\, Q'))$$

by applying rule 1.1.

A.3 Communication compositions

In this section three common situations of communicating processes are discussed as shown in figure A.3. We will shown that these three situations can be described by the same description for three different compositions towards process *P*. The arrows *a*, *b*, and *c*, are channels over which the processes *P*, *Q*, *R*, and *S* pass information. This information will be passed according to the rendezvous principle. The channels are one-way which direction is denoted by the arrows.

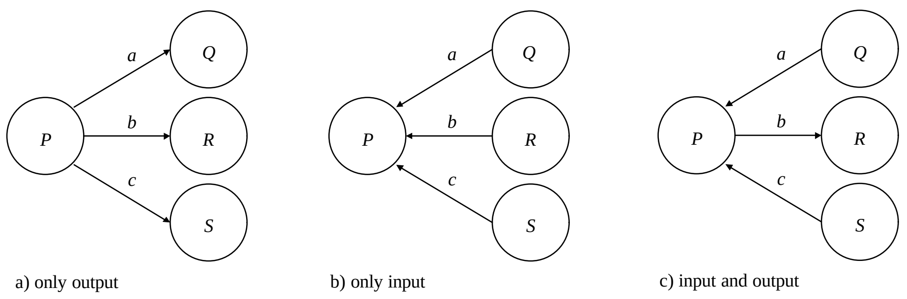


Figure A.3: Three situations towards process P

Here, we consider process *P* as primary process and the others as secondary processes. These situations show the communication network of a program and the degree of parallelism i.e. the structure and geometric parallelism. However, they tell nothing about the order and time of which communication takes pace. The order of communication by process *P* will be discussed for three different compositions. Time aspects will not be discussed here.

The secondary processes only communicate with process *P* and there is no mutual communication between the secondary processes themselves. The system process can be described as

$$SYSTEM \quad = \quad P \,|[\alpha P \mid \alpha Q \mid \alpha R \mid \alpha S \,]|\, (Q \,|||\, R \,|||\, S) \qquad (*)$$

$$\text{with} \quad \alpha P = \{a,b,c\},\ \alpha Q = \{a\},\ \alpha R = \{b\},\ \alpha S = \{c\}.$$

Here each process behaves according to its own definition, but with the constraint that events which are in the alphabet of both *P* and ($Q \,|||\, R \,|||\, S$), i.e. events *a*, *b*, and *c*, require their simultaneous participation. However they may process independently on those events solely to their own alphabet i.e. processes *Q*, *R*, and *S* process independently. We may write the set $\{\alpha P \mid \alpha Q \mid \alpha R \mid \alpha S\}$ as $\{\{a,b,c\}|\{a\}|\{b\}|\{c\}\}$ or as $\{a,b,c\}$ to simplify the equations.

The primary process *P* can communicate with the secondary processes *Q*, *R*, and *S*, in a sequential order, in parallel, or by choice. Therefor, we can consider respectively three compositions towards communication; (i) the sequential composition, (ii) the parallel composition, and (iii) the alternative composition. In this section we will discuss these composition towards figure A.3.

A.3.1 Sequential composition

A sequential composition performs communication in a certain sequential order. In case process *P* is a thread of control, i.e. sequential flow of control, this process is sequential by nature. The programmer could specify a certain order as defined by

$$\begin{aligned} P &= (a \rightarrow b \rightarrow c \rightarrow P') \\ Q &= (a \rightarrow Q') \\ R &= (b \rightarrow R') \\ S &= (c \rightarrow S') \end{aligned}$$

where the processes *P'*, *Q'*, *R'* and *S'* are suffix processes after the composition.

After substituting these behaviors into the system equation (*) the sequential composition can be described as

$$SEQ = (a \rightarrow b \rightarrow c \rightarrow P') \,|[\{a,b,c\}]|\, (\, (a \rightarrow Q') \,|||\, (b \rightarrow R') \,|||\, (c \rightarrow S')\,)$$

This definition is the same for all three situations of figure A.3. The sequence is in its entirely determined by process *P*. The direction of communication is not relevant.

A.3.2 Parallel composition

A parallel composition performs communication in parallel. If the order of communication by process *P* is undefined then a parallel composition should be used to avoid deadlock at any time. The behavior of the process *P* in which it wants to communicate simultaneously can be defined as

$$\begin{aligned} P &= (a \rightarrow b \rightarrow c \rightarrow P') \square (a \rightarrow c \rightarrow b \rightarrow P') \square (b \rightarrow a \rightarrow c \rightarrow P') \\ &\quad \square (b \rightarrow c \rightarrow a \rightarrow P') \square (c \rightarrow a \rightarrow b \rightarrow P') \square (c \rightarrow b \rightarrow a \rightarrow P') \\ Q &= (a \rightarrow Q') \\ R &= (b \rightarrow R') \\ S &= (c \rightarrow S') \end{aligned}$$

The entire equation is

$$\begin{aligned} PAR &= (\, (a \rightarrow b \rightarrow c \rightarrow P') \square (a \rightarrow c \rightarrow b \rightarrow P') \square (b \rightarrow a \rightarrow c \rightarrow P') \\ &\quad \square (b \rightarrow c \rightarrow a \rightarrow P') \square (c \rightarrow a \rightarrow b \rightarrow P') \square (c \rightarrow b \rightarrow a \rightarrow P')\,) \\ &\quad |[\{a,b,c\}]|\, (\, (a \rightarrow Q') \,|||\, (b \rightarrow R') \,|||\, (c \rightarrow S')\,) \end{aligned}$$

The complexity of the *PAR* equation is of magnitude O(*n*!), in which *n* is the number of parallel channels out of *P*.

In practice, simulating this approach by `if-then-else` statements will be enormous complex. However, this process can be simplified by using sub-processes P_i, for each channel input or output.

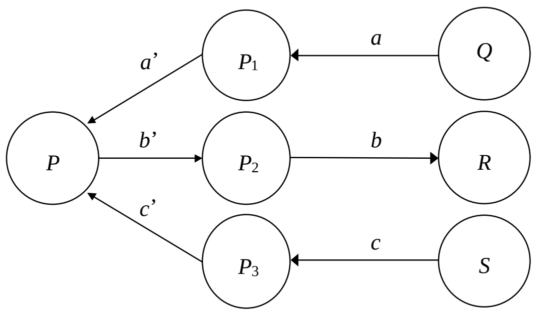


Figure A.4: Parallel composition of situation c.

The sub-processes P_1, P_2, and P_3 provide communication in parallel. Process P can execute the read and write actions in a arbitrary sequential order which will actually be processed in parallel by the sub-processes. The suffix process P' may only continue when all the sub-processes are finished.

Process P can be defined similar as the *SEQ* process of section A.3. The events a', b', and c' are shared events that indicate, when engaged in, that process P has started its sub-processes P_i. Some additional $join_{P1}$, $join_{P2}$, and $join_{P3}$ events indicate, when engaged in, that the respectively process P_1, P_2, and P_3 have finished processing its task.

Let

$$\begin{aligned}
P &= (a' \rightarrow b' \rightarrow c' \rightarrow join_{P1} \rightarrow join_{P2} \rightarrow join_{P3} \rightarrow P') \,|[\{a',b',c'\}]|\, (P_1 \,|||\, P_2 \,|||\, P_3) \\
P_1 &= (a' \rightarrow a \rightarrow join_{P1} \rightarrow P_1) \\
P_2 &= (b' \rightarrow b \rightarrow join_{P2} \rightarrow P_2) \\
P_3 &= (c' \rightarrow c \rightarrow join_{P3} \rightarrow P_3) \\
Q &= (a \rightarrow Q') \\
R &= (b \rightarrow R') \\
S &= (c \rightarrow S')
\end{aligned}$$

$$\begin{aligned}
P_1 \,|[\{a',a,join_{p1}\}]|\, Q &= (a' \rightarrow a \rightarrow join_{P1} \rightarrow (P_1 \| Q')) \\
P_2 \,|[\{b',b,join_{p2}\}]|\, R &= (b' \rightarrow b \rightarrow join_{P2} \rightarrow (P_2 \| R')) \\
P_3 \,|[\{c',c,join_{p3}\}]|\, S &= (c' \rightarrow c \rightarrow join_{P3} \rightarrow (P_3 \| S'))
\end{aligned}$$

Process P triggers the sub-processes P_1, P_2, and P_3 through the events a', b' and c' and subsequently these sub-processes trigger a common join through the events $join_{P1}$, $join_{P2}$, and $join_{P3}$.

$$\begin{aligned}
PAR &= (a' \rightarrow b' \rightarrow c' \rightarrow join_{P1} \rightarrow join_{P2} \rightarrow join_{P3} \rightarrow P') \,|[\{a',b',c'\}|\{a'\}|\{b'\}|\{c'\}]| \\
&\quad ((P_1 \,|[\{a',a\}|\{a\}]|\, Q) \,|||\, (P_2 \,|[\{b',b\}|\{b\}]|\, R) \,|||\, (P_3 \,|[\{c',c\}|\{c\}]|\, S)) \\
&= (a' \rightarrow b' \rightarrow c' \rightarrow join_{P1} \rightarrow join_{P2} \rightarrow join_{P3} \rightarrow P') \,|[\{a',b',c'\}|\{a'\}|\{b'\}|\{c'\}]| \\
&\quad ((a' \rightarrow a \rightarrow join_{P1} \rightarrow (P_1 \| Q')) \,|||\, (b' \rightarrow b \rightarrow join_{P2} \rightarrow (P_2 \| R')) \\
&\quad \,|||\, (c' \rightarrow c \rightarrow join_{P3} \rightarrow (P_3 \| S')))
\end{aligned}$$

The complexity of the *PAR* equation is of magnitude O(n+n), by which n is the number of parallel channels out of P. By a handy trick in the implementation (not in CSP) we can reduce the complexity to O(n+1) by using one join.

A.3.3 Alternative composition

A simple alternative composition based on one event can be described by the process

$$ALT \quad = \quad (a \rightarrow P) \,\square\, (\neg a \rightarrow Q)$$

The *ALT* will behave as *P* if it can engage in *a* else it will behave as *Q*. This behavior continues immediately and is analogous to a simple `if-then-else` statement. The process for the `if-then` statement – without the `else` – can be described as

$$ALT \quad = \quad (a \rightarrow P) \,\square\, (\neg a \rightarrow SKIP)$$

The next expression describes an alternative composition of figure A.3. Process *P* makes choices based on the events *a*, *b*, and *c* as described as

$$\begin{aligned} P \quad &= \quad (a \rightarrow P_a') \,\square\, (b \rightarrow P_b') \,\square\, (c \rightarrow P_c') \\ Q \quad &= \quad (a \rightarrow Q') \\ R \quad &= \quad (b \rightarrow R') \\ S \quad &= \quad (c \rightarrow S') \end{aligned}$$

The entire equation is

$$\begin{aligned} ALT \quad = \quad & ((a \rightarrow P_a') \,\square\, (b \rightarrow P_b') \,\square\, (c \rightarrow P_c')) \,|[\{a,b,c\}|\{a\}|\{b\}|\{c\}]|\, \\ & ((a \rightarrow Q') \,|||\, (b \rightarrow R') \,|||\, (c \rightarrow S')) \end{aligned}$$

If more than one event is true then the choice will be non-deterministic. The choice can be made by a fair or by a priority mechanism. The events *a*, *b*, and *c* are called *guards* of the alternative composition. A guard can be input or output guarded.

The complexity of the *ALT* equation is of magnitude O(*n*), by which *n* is the number of parallel channels out of *P*.