Java Communicating Sequential Processes

'Design Of JCSP Language Classes'

Paul Austin
pda1@ukc.ac.uk
University Of Kent Canterbury
BSc Computer Science with an Industrial Year
3rd Year Project

Contents

1 Introduction 1

2 Functionality 2

2.1 Channels 2
2.1.1 Cardinality 3
2.1.1.1 One to One Channel 4
2.1.1.1.1 Reading 4
2.1.1.1.2 Writing 4
2.1.1.2 One To Many Channel 5
2.1.1.2.1 Reading 5
2.1.1.2.2 Writing 5
2.1.1.3 Many To One Channel 6
2.1.1.3.1 Reading 6
2.1.1.3.2 Writing 6
2.1.1.4 Many To Many Channel 7
2.1.1.4.1 Reading 7
2.1.1.4.2 Writing 7
2.1.2 Storage Types 8
2.1.2.1 Single Object 8
2.1.2.2 Buffer 8
2.1.2.3 Over Writing buffer 8
2.1.2.4 Timer 8

2.2 Alternative construct 8

2.3 Parallel Construct and Process's 10

3 Design & Implementation 11

3.1 Channels 11
3.1.1 Interface 11
3.1.1.1 Reading and writing 11
3.1.2 Implementation 15
3.1.2.1 Inheritance Based Channels 15
3.1.2.2 Bridge Based Channels 16
3.1.2.2.1 ObjectStore 17
3.1.2.2.1.1Single Object 18
3.1.2.2.1.2Buffer 18
3.1.2.2.1.3OverwritingBuffer 19
3.1.2.2.1.4Timer 19
3.1.2.2.2 The read method 20
3.1.2.2.3 The write method 21

3.2 Alternative Construct 22
3.2.1 Interface 22
3.2.1.1 Alternative 22
3.2.1.2 Alting Channels 24
3.2.2 Implementation 26
3.2.2.1 Alternative 26
3.2.2.2 AltingChannel 27

3.3 Parallel Construct and Processes 29
3.3.1 Interface 29
3.3.1.1 Using Java Threads 29
3.3.1.2 OCCAM style PAR Construct 31
3.3.1.2.1 CSProcess 31
3.3.1.2.2 Parallel 32

3.3.1.2.3 ProcessNetwork 34
3.3.2 Implementation........ 35
3.3.2.1 Using Java Threads........ 35
3.3.2.2 OCCAM style PAR Construct........ 35
3.3.2.2.1 CSProcess 36
3.3.2.2.2 Parallel 36
3.3.2.2.3 ProcessNetwork 39

Appendix A Class Diagrams........ 40

A.1 JCSP Language Classes Overview 40

A.2 JCSP Language Classes Detail 41
A.2.1 Channel Classes........ 41
A.2.2 Parallel Classes........ 42

Figures

Diagrams

Figure 1 - Relationship between Channel interfaces..........12
Figure 2 - JCSP Channel interface relationships..........14
Figure 3 - Inheritance based Channels - Class Diagram..........15
Figure 4 - Bridge Channels - Class Diagram..........17
Figure 5 – SingleObject - State Diagram..........18
Figure 6 – Buffer - State Diagram..........18
Figure 7 – OverWritingBuffer - State Diagram..........19
Figure 8 – Timer - State Diagram..........19
Figure 9 - Class interactions for the Channel read method..........20
Figure 10 - Class interactions for the Channel write method..........21
Figure 11 – JCSP Channel interface relationships with Alting..........25
Figure 12 – JCSP Channel interfaces with Alting..........25
Figure 13 - Alternative - select..........27
Figure 14 - AltingChannel - write..........28
Figure 15 - Relationship between Parallel and CSProcess..........32
Figure 16 - Relationship between ProcessNetwork and CSProcess..........35
Figure 17 - Parallel Class Diagram..........36
Figure 18 - Parallel – run()..........37
Figure 19 - Parallel State Diagram..........39
Figure 20 - JCSP Language classes – Overview..........40
Figure 21 - JCSP Channel classes - Detail..........41
Figure 22 - JCSP Parallel classes - Detail..........42

Code Fragments

Code 1 - One to One Channel, reading Pseudo Code..........4
Code 2 - One to One Channel, writing Pseudo Code..........4
Code 3 - One to Many Channel, reading Pseudo Code..........5
Code 4 - One to Many Channel, writing Pseudo Code..........5
Code 5 - Many to One Channel, reading Pseudo Code..........6
Code 6 - Many to One Channel, writing Pseudo Code..........6
Code 7 - Many to Many Channel, reading Pseudo Code..........7
Code 8 - Many to Many Channel, writing Pseudo Code..........7
Code 9 - ALT process in OCCAM..........9
Code 10 - Using Channel Interfaces - example..........12
Code 11 - JTW Channel interface..........13
Code 12 - CJT Channel interface..........13
Code 13 - JCSP Channel interface..........14
Code 14 - ObjectStore interface..........17
Code 15 - Alternative class - public interface..........23
Code 16 – Alternative interface - example..........23
Code 17 - AltingChannel - interfaces..........24
Code 18 - Algorithm for selection of Channels..........26
Code 19 – Parallelism using threads..........30
Code 20 – CSProcess interface..........31
Code 21 – CSProcess example..........32
Code 22 – Parallel interface..........33
Code 23 – Parallel Example..........34
Code 24 – ProcessNetwork interface..........35
Code 25 – Parallel run() method – pseudo code..........36

1 Introduction

This document describes the functionality and design of the classes in the Java Communicating Sequential Processes package that are used to model the OCCAM language in Java. The design will be described using the Unified Modelling Language [1] (UML) notation with descriptive text describing why certain decisions were made. The discussion will assume knowledge of the facilities and concepts of the CSP [2] model in the OCCAM [3] language and of programming in the Java [4] programming language.

Several different versions of CSP style channels have been developed at the University of Kent England and the University of Twente Netherlands. Each different version provides slightly different interfaces to channels. Included in these packages are some other classes that provide different types of channel, composition constructs (Parallel and Alternative) and standard building blocks.

In this document, the following terms will be used; 'Java Threads workshop' (JTW) [5][6] for the work at the University of Kent, 'Communicating Java Threads' (CJT) [7] for the work at the University of Twente and 'Java Communicating Sequential Processes' (JCSP) for the work described in this document.

The package described within this document takes the best parts from each of these versions to develop a unified class library. The library will try to contain classes that are simple to use for developers constructing applications and is simple to extend to include extra channel types.

The design of these channels must balance the need to provide easily extensible channels that have as low a performance overhead as possible.

2 Functionality

This section describes the functionality that the library must implement to provide features similar to those offered by OCCAM.

2.1 Channels

A channel provides a mechanism for two (or more) parallel processes to communicate in a synchronised manner. Therefore the process performing output cannot continue until the channel accepts the read request and likewise the process performing input cannot continue until the channel accepts the write request. This is specifically a generic description of a channel so that in our class libraries we can define other channels that implement the behaviour of accepting read and write processes in different ways that still follow this model.

The OCCAM language defines a Channel to be synchronised, point to point, non-buffered communication. In this definition, we have only one reader and one writer (point to point). When the writer writes to the Channel, it cannot continue execution until the reader has read from the Channel (synchronised). The Channel does not logically store the value of the object in the communication (non-buffered).

In the Java version, we want to build upon this basic concept of a channel to include other type of channel such as; Timer, io ports, Buffered Channels, Overwriting Buffers, Shared Channels and Network channels. In fact in OCCAM 3 there are Shared Channels. It is possible with the basic channel to build active processes that implement Buffered Channels, Overwriting Buffers etc. It is however sometimes more useful to have low level versions of these type of channels (for example when a parallel version incurs too much overhead).

In OCCAM, the channel is implemented as a primitive type in the language. This provides the benefits of ensuring at compile time that only one process is permitted to write to the channel at a time (for One to One channels). In addition, the types of data sent down the channel are specified for the channel and this is check at compile time. To check that only one process could write to a channel in Java run time checks would be required which would add an extra overhead and possibly require a more complex interface. It is therefore the responsibility of the software constructor to decide on the type of channel to use.

It is possible in Java to implement channels, which can only accept certain types of data. Unfortunately, as the language does not support template classes this would have to be implemented using run time checks or by creating new a *`Channel`* class for each type of *`Object`* that is to be sent down a channel. This would either result in a performance hit or a very large class library. In this class library any *`Object`* can be sent down the channel with no checks made by the library as to the type of the data. It is therefore the responsibility of the users of these channels to publicise the data types expected down a particular channel and to handle any error conditions.

The OCCAM channels make a copy of the data sent down a channel from the original process to the receiving process. In the Java channels this will not be done automatically by the channel, instead it is the responsibility of the developer to copy *`Object`* if required. The reasons for not copying the *`Object`* are listed below.

1) A lot of classes in the Java library are immutable[1] therefore creating a copy is not necessary as it safe for more than one thread to access the *`Object`*.
2) Copying an *`Object`* in Java is performed using the `clone()` method on the *`Object`* which is a protected method therefore it cannot be called unless the actual class has declared the `clone()` method to be public. To finally copy the *`Object`* it is necessary to cast the reference to the actual class and then call the `clone()` method. This would require some fairly complex code or require a common super class for all types that should be cloned that defines a `clone()` to be public.
3) It would be possible to use serialisation to make copies of the *`Object`* sent down the *`Channel`*, this would however have a large performance hit.

2.1.1 Cardinality

The cardinality of a channel defines how many different processes can be trying to either read or write at the same time. It is necessary to consider the cardinality of the channels instead of implementing the channels as a ‘Many to Many’ channel. There are several reasons why this is so.

1) If the class library includes support for Alternative selection of channels it is only safe to do this for a single reader (see section 2.3 for further details).
2) The ‘Many to Many’ channel has an extra synchronisation for each read and write to ensure the channel is left in a consistent state and does not cause any race hazards. This extra synchronisation is necessary for the ‘Many to Many’ channel as if it did not have this extra synchronisation the following scenario could arise. A second writer could enter the write method and overwrite the data from the first process, while the first process was blocked waiting for a reader to read the data. For channels that only allow one reader (or writer) this extra synchronisation on reading from (or writing to) the channel adds an unnecessary overhead to each channel communication. This overhead is quite high as the Java synchronization mechanism takes a long time.

There are four cardinalities to consider; each one is discussed below.

4) Single Writer and Single Reader (*`One2OneChannel`*)
1. Single Writer and Many Readers (*`One2ManyChannel`*)
2. Many Writers and Single Reader (*`Many2OneChannel`*)
3. Many Writers and Many Readers (*`Many2ManyChannel`*)

[1] immutable means the value of the *`Object`* cannot be changed once the *`Object`* has been created. An example is the *`String`* class.

2.1.1.1 One to One Channel

The simplest form of channel is the One To One channel, that is only safe to be used for one reader and writer. This is the type of channel used in OCCAM.

2.1.1.1.1 Reading

Reading from the channel involves obtaining the monitor for the channel to ensure exclusive access. A check is then made to see if there is data available, if there is no data the monitor is released and this process will wait for a writer. The *`Object`* will be read from the data store and the waiting writer will be notified. The *`Object`* is returned and the monitor is released.

```
synchronize on channel
  if data store is empty
    wait for writer
  end if
  read Object from data store
  notify waiting writer
  return Object
end synchronize
```

Code 1 - One to One Channel, reading Pseudo Code

2.1.1.1.2 Writing

Reading from the channel involves obtaining the monitor for the channel to ensure exclusive access. The *`Object`* will be written to the data store and the waiting reader will be notified. A check is then made to see if the data store is full, if it is full the monitor is released and this process will wait for a reader. The monitor is then released.

```
synchronize on channel
  write Object to data
store
  notify waiting reader
  if data store is full
    wait for reader
  end if
end synchronize
```

Code 2 - One to One Channel, writing Pseudo Code

2.1.1.2 One To Many Channel

The One to Many Channel is safe to be used for many readers and one writer.

2.1.1.2.1 Reading

Reading from the channel involves obtaining the monitor for read and then for the channel to ensure exclusive access. A check is then made to see if there is data available, if there is no data the monitor is released and this process will wait for a writer. The *`Object`* will be read from the data store and the waiting writer will be notified. The *`Object`* is returned and the monitor is released.

```
synchronize on read monitor
  synchronize on channel
    if data store is empty
      wait for writer
    end if
    read Object from data store
    notify waiting writer
    return Object
  end synchronize
end synchronize
```

Code 3 - One to Many Channel, reading Pseudo Code

2.1.1.2.2 Writing

Reading from the channel involves obtaining the monitor for the channel to ensure exclusive access. The *`Object`* will be written to the data store and the waiting reader will be notified. A check is then made to see if the data store is full, if it is full the monitor is released and this process will wait for a reader. The monitor is then released.

```
synchronize on channel
  write Object to data
store
  notify waiting reader
  if data store is full
    wait for reader
  end if
end synchronize
```

Code 4 - One to Many Channel, writing Pseudo Code

2.1.1.3 Many To One Channel

The One to Many Channel is safe to be used for one reader and many writers.

2.1.1.3.1 Reading

Reading from the channel involves obtaining the monitor for the channel to ensure exclusive access. A check is then made to see if there is data available, if there is no data the monitor is released and this process will wait for a writer. The *`Object`* will be read from the data store and the waiting writer will be notified. The *`Object`* is returned and the monitor is released.

```
synchronize on channel
  if data store is empty
    wait for writer
  end if
  read Object from data store
  notify waiting writer
  return Object
end synchronize
```

Code 5 - Many to One Channel, reading Pseudo Code

2.1.1.3.2 Writing

Reading from the channel involves obtaining the monitor for write and then for the channel to ensure exclusive access. The *`Object`* will be written to the data store and the waiting reader will be notified. A check is then made to see if the data store is full, if it is full the monitor is released and this process will wait for a reader. The monitor is then released.

```
synchronize on write monitor
  synchronize on channel
    write Object to data store
    notify waiting reader
    if data store is full
      wait for reader
    end if
  end synchronize
end synchronize
```

Code 6 - Many to One Channel, writing Pseudo Code

2.1.1.4 Many To Many Channel

The One to Many Channel is safe to be used for many readers and many writers.

2.1.1.4.1 Reading

Reading from the channel involves obtaining the monitor for read and then for the channel to ensure exclusive access. A check is then made to see if there is data available, if there is no data the monitor is released and this process will wait for a writer. The *`Object`* will be read from the data store and the waiting writer will be notified. The *`Object`* is returned and the monitor is released.

```
synchronize on read monitor
  synchronize on channel
    if data store is empty
      wait for writer
    end if
    read Object from data store
    notify waiting writer
    return Object
  end synchronize
end synchronize
```

Code 7 - Many to Many Channel, reading Pseudo Code

2.1.1.4.2 Writing

Reading from the channel involves obtaining the monitor for write and then for the channel to ensure exclusive access. The *`Object`* will be written to the data store and the waiting reader will be notified. A check is then made to see if the data store is full, if it is full the monitor is released and this process will wait for a reader. The monitor is then released.

```
synchronize on write monitor
  synchronize on channel
    write Object to data store
    notify waiting reader
    if data store is full
      wait for reader
    end if
  end synchronize
end synchronize
```

Code 8 - Many to Many Channel, writing Pseudo Code

2.1.2 Storage Types

The storage types define how objects sent across the channel are stored. The storage policy may specify the number of `Objects` that can be in the Channel and what to do if it is full. The following storage types will be provided in the JCSP library; `SingleObject`, `Buffer`, `OverWritingBuffer` and `Timer`. Further `ObjectStores` can be added by developers.

2.1.2.1 Single Object

The OCCAM channels store only one value at a time, this value is not buffered and the writer must block until a reader has read the value and a reader must block until data has been written. In the library, the `SingleObject` will provide this functionality.

2.1.2.2 Buffer

A `Buffer` acts in a very similar way to the rendezvous channel except that a `Buffer` can hold more than one `Object` at a time. If the `Buffer` is empty, the reader must block until a further `Object` is written. If the `Buffer` is full, the writer must wait until a reader reads an `Object` from the channel.

2.1.2.3 Over Writing buffer

An `OverWritingBuffer` has the same properties as a `Buffer` but instead of blocking when the buffer is full the `OverWritingBuffer` discards the last value written to the buffer and writes the new value in it's place.

2.1.2.4 Timer

OCCAM provides a special type of variable called a Timer that can be read from as if it were a Channel. Reading from Timer returns a long representing the current time, the Timer never blocks the caller. Writing to a Timer is not possible in OCCAM, the JCSP `Timer ObjectStore` will discard any data written to it.

2.2 Alternative construct

An Alternation combines several input, timeout or `SKIP` clauses, only one of which is executed. Each clause may optionally have a Boolean guard; the input may only be executed if the guard is true. Associated with each clause is a section of code (process) which is executed if that clause is selected. The input clause is a read from a single channel that is ready if data is available to be read from the channel. The timeout clause that will wait until the time specified has passed; the clause will become ready after this time. The `SKIP` clause is always ready and will be selected if none of the input clauses is immediately ready.

When the Alternation construct is executed the channels specified will be probed to see if it has data to be read. The following checks will be executed in the order shown below.

1. If only one channel is ready the data will be read from it and the associated code will be executed.

2. If more than one channel is ready, one of these will be selected. The data will be read from it and the associated code will be executed (there is no specification about which channel will be selected).
3. If there is a `SKIP` clause the Alternation will not read from any channels and the code associated with `SKIP` will be executed.
4. If no channels are ready the Alternation will wait until at least one of the channels becomes ready
5. An Alternation may also have a timeout.

The OCCAM language provides the facility to wait for input on more than one channel and then select one of the channels which has data ready to be read. The data from the selected channel is read and then some code will be executed to process the data read. The OCCAM code fragment shows an example of an `ALT` statement that waits for input on the three channels in1, in2 and in3. When input is available the `ALT` process will select one of the channels which has data and executes the read from that channel and the code associated with that read.

```
BOOL b:
TIMER tim:
SEQ
  B := false
  ALT
    INT i:
    in1 ? i
      SEQ
        out ! 1
        out ! i
        b := true
    b & in2 ? i
      SEQ
        out ! 2
        out ! i
    tim ? AFTER 10
      SEQ
        out ! 3
        out ! I
    b & SKIP
      SEQ
        out ! 0
        out ! 0
```

Code 9 - ALT process in OCCAM

2.3 Parallel Construct and Process's

The `PAR` construct in OCCAM enables the parallel execution of fine grained processes such as a single assignment, output to a channel or read from a channel as well as named procedures (`PROCs`). The `PAR` construct after setting the processes running will not complete until all the specified processes terminate.

The language also enforces certain rules about parallelism to ensure that no two processes try to write/read to/from the same channel or modify the same variable. These constraints mean that it is possible to reason about the programs, as the language does not allow code to be written with undesirable side effects.

The Java version of `PAR` must provide the following;
Execution of processes in parallel as per `PAR`.

- Encapsulation of processes in named procedures
- Must not terminate until all sub processes terminate

In addition, it would be useful if it could provide the following;

- Addition of new Processes
- Removal of Processes
- Asynchronous execution of a Process

3 Design & Implementation

The section discusses the different designs for the channels considered comparing their relative merits along with the final design used. The main factors considered were (in no particular order).

- Good OO design (high cohesion, low coupling and encapsulation)
- Simplicity of interface
- Performance

For the different parts of the library, first the interface to the classes is described then followed by a section talking about the different implementations tried and the relative merits.

The design of the library was considered in the following groups of classes.

1. Channels
2. Alternative
3. Parallel construct and Processes

3.1 Channels

In designing the `Channel` classes it is important, to design the classes is such a way that it is easy to add new types of channel. The following sections describe the approach taken to allow this to happen.

3.1.1 Interface

To enable the facility to change the implementation of channels and have more than one type of `Channel` the interface of the channels were designed as pure interfaces with no implementation. This enables clients to be written without any knowledge of the actual `Channel` it uses.

3.1.1.1 Reading and writing

When building CSP processes it is usually the case that a particular process will use a `Channel` either for input or for output but not both input and output. To aid the designer of these processes it should be possible for them to specify that a particular `Channel` is for either for input or output, the compiler would then flag an error if the read method was called on an output `Channel`.

The original JTW channels, had only one type of `Channel` used to send arbitary `Objects`, there was no mechanism for processes to specify if a `Channel` is to be used for input or output. The CJT channels included two interfaces[2] types for input and output respectively. The `ChannelInput` interface defines the format of the `read()` method and the `ChannelOutput` interface defines the format of the `write()` method. The `Channel` interface is then defined as implementing both the `ChannelInput` and `ChannelOutput` interfaces as shown in the class diagram below. (`Channel` could actually be defined as a class which implements the `read()` and `write()`methods as in the CJT classes, in the JCSP classes it is just going to be implemented as an interface).

[2] Interfaces in Java are used to specify the public methods a class must implement to have the type specified by the interface. A class may implement several interfaces as well as inheriting from one class. As Java does not allow multiple inheritance the inclusion of interfaces enables Objects to have multiple types.

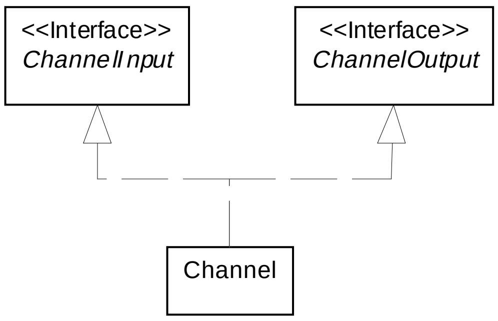
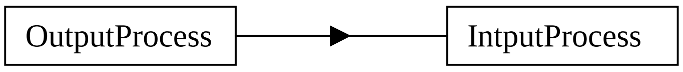


Figure 1 - Relationship between Channel interfaces

The example below shows how processes would use the interfaces to access the channel, the example network consists of one process outputing along the channel and one reading from it. The definition of the *`OutputProcess`* uses an *`ChannelOutput`* reference in the constructor, this means that inside the class only those methods defined in the *`ChannelOutput`* interface can be used and this restraint is checked at compile time. The same is true for the *`InputProcess`* except only those methods defined in *`ChannelInput`* may be used. As the *`Channel`* interface implements the type of both *`ChannelInput`* and *`ChannelOutput`* it is legal for a *`Channel`* to be passed where a *`ChannelInput`* or *`ChannelOutput`* is required.



```
class Network {
  Network() {
    Channel c = new Channel();
    OutputProcess writer = new OutputProcess(c);
    InputProcess reader = new InputProcess(c);
  }
}

class OutputProcess {
  OutputProcess(ChannelOutput out) {
    //perform initialisation here
  }
  public void run() {
  }
}

class InputProcess {
  InputProcess(ChannelInput in) {
    //perform initialisation here
  }
  public void run() {
  }
}
```

Code 10 - Using Channel Interfaces - example

After considering the relationships between the interfaces it is necessary to define the methods required to read and write data to and from the channel. The format of these methods should be as intuitative and simple as possible to use and provide flexibility in the type of data sent down the channel.

The JTW channels defined the `write()` method to accept an *`Object`* as a parameter and the `read()` method to return an *`Object`*. The channels do not copy the *`Object`* as they are sent across the channel. The definition of the class is shown below.

```
public class Channel {
  public void write(Object o) { … }
  public Object read() { … }
  …
}
```

Code 11 - JTW Channel interface

The CJT classes provide a more complex interface to reading and writing that includes support for alternative selection of channels and parrallel input/output. The CJT channels can only send data which is a subclass of *`ClonableProtocol`* which makes the interface less flexibile. The channels do however copy the *`Object`* as it is sent across the channel. The definition of the class is shown below.

```
public class Channel {
 public void write(ClonableProtocol o) throws Exception { … }
 public void write(ALT alt, ClonableProtocol o) throws Exception {
… }
 public void write(PAR par, ClonableProtocol o) throws Exception {
… }

 public void read(ClonableProtocol o) throws Exception { … }
 public void read(ALT alt, ClonableProtocol o) throws Exception {
… }
 public void read(PAR par, ClonableProtocol o) throws Exception {
… }
  …
}
```

Code 12 - CJT Channel interface

There are several issues with the interface, which are;

- It is only possible to send *`ClonableProtocol`* objects down the channel. The reason for this is to enable the *`Object`* to be copied when passed across the channel. As the JCSP channels are required to send an arbitrary *`Object`* and therefor it is the client that is responsible to copy the *`Object`* if required. The inflexibility of the CJT interface is not necessary and causes the following problem.
- The *`ClonableProtocol`* class defines the clone method as `clone(Object)` instead of `Object clone()`. This would cause a *`ClassCastException`* if the variable the client is trying to read into is not the same as the variable sent down the channel.

- The read method is not intuitive.
- All the read and write methods are declared to throw all exceptions, this forces the implementers of processes to place a try catch block around the calls to `read()` and `write()`. The only time exceptions would be raised by the channels are *`RuntimeExceptions`* which signify program errors and should not be caught.
- The `read()`and `write()`methods provide a way to poll a Channel to see if communication is possible using the ALT construct. This is an inefficient method and providing alting on both input and output causes problems if both the sender and receiver are alting on the same channel. The implementation of `ALT` in OCCAM does not allow this for the same reasons (see 3.1.2 for a further discussion of `ALT`).

- The JCSP library will use the method of having *`ChannelInput`* and *`ChannelOutput`* interfaces which define the `read()`and `write()` methods respecivley as per the CJT classes. The format of the read and write methods are the same as the JTW classes as they are simple and intuitive. The `read()`and `write()` methods for `PAR` and `ALT` provided by CJT are not necessary as these will be implemented differently. The definitions of these interfaces are shown below and the class relationships are shown in Figure 2.

```
public interface ChannelOutput {
  public void write(Object o);
}

public interface ChannelInput {
  public Object read();
}

public interface Channel extends ChannelOutput,
ChannelInput {
}
```

Code 13 - JCSP Channel interface

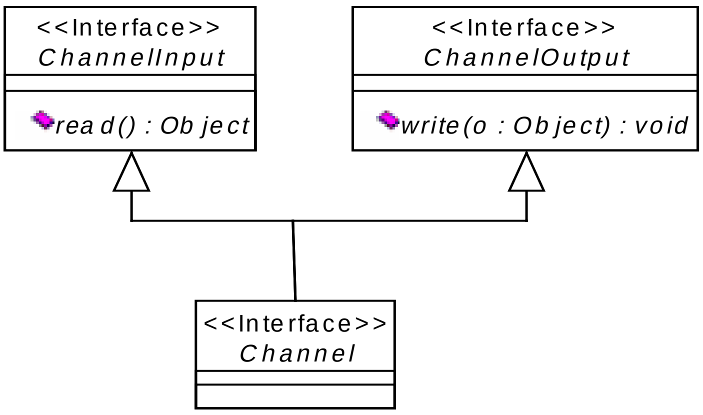


Figure 2 - JCSP Channel interface relationships

3.1.2 Implementation

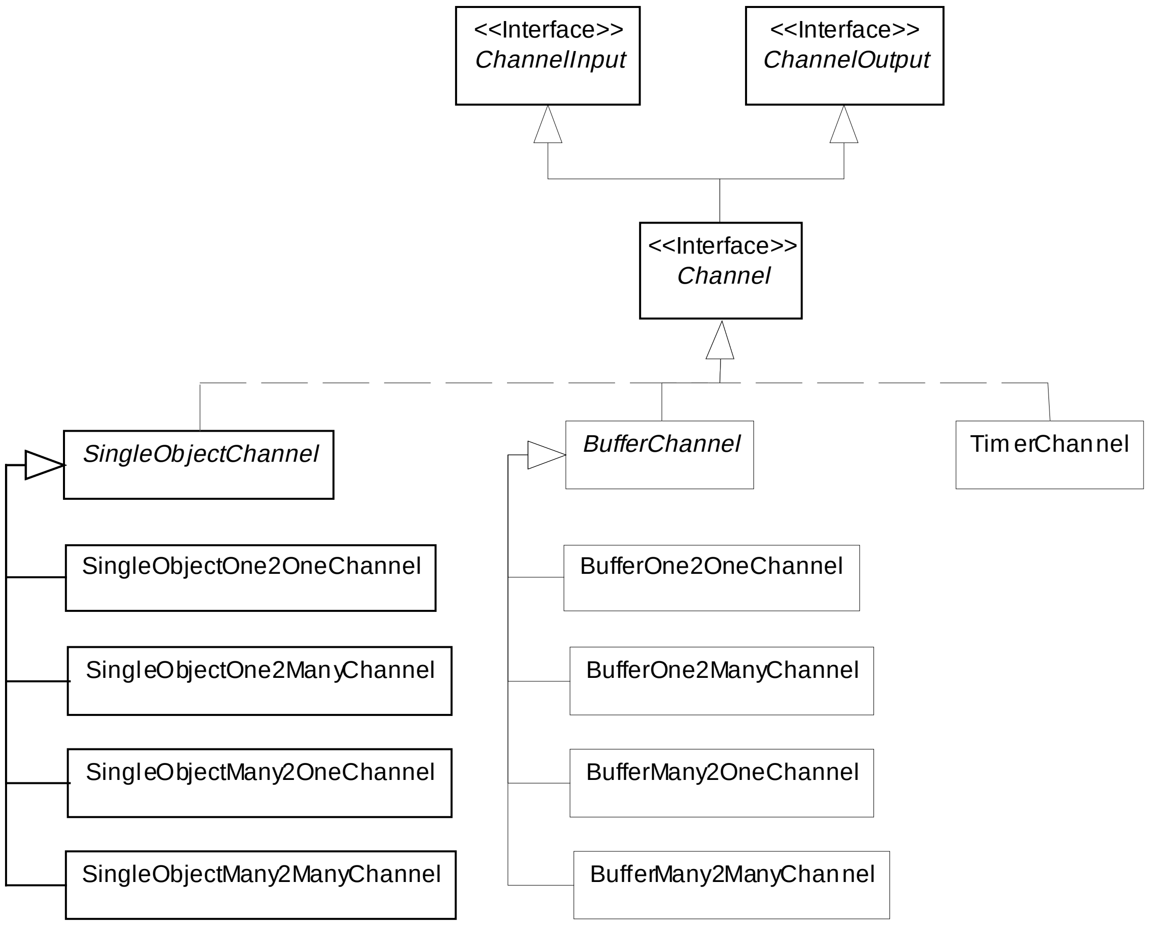
For the implementation of the channels two different methods were considered, each of these is discussed below

1. Inheritance based channels
2. Bridge based channels

3.1.2.1 Inheritance Based Channels

The first approach considered when designing the implementation for the channels was to take the interface and create a class for each of the storage types that implements *`Channel`*. Each of these classes would then define the `read()` and `write()` methods to implement the storage policy as well as the synchronisation between the reading and writing processes.

The Class Diagram below shows the relationships between the classes in the inheritance based model.



<<Interface>> ChannelInput
<<Interface>> ChannelOutput
<<Interface>> Channel
SingleObjectChannel
BufferChannel
TimerChannel
SingleObjectOne2OneChannel
SingleObjectOne2ManyChannel
SingleObjectMany2OneChannel
SingleObjectMany2ManyChannel
BufferOne2OneChannel
BufferOne2ManyChannel
BufferMany2OneChannel
BufferMany2ManyChannel

Figure 3 - Inheritance based Channels - Class Diagram

To implement the cardinality of the channels up to four sub classes of each of the channels need to be created to make channels that are safe to use for many readers and writers. The four cardinalities required are One2One, One2Many, Many2One and Many2Many. Certain channels such as the *`TimerChannel`* may not need to define extra classes to make them safe to use for many readers and writers.

Note: For each new storage policy five classes are required and the channel synchronisation code will have to be written for each channel type.

This is one method of implementing the Inheritance model for the channels, another approach may have the cardinalities at the top of the tree and the storage types subclass these. Whatever way the inheritance is organised the number of classes required is large.

Another problem with this approach is if network Channels were introduced, it would be necessary to create *`BufferNetworkChannel`*, *`SingleObjectNetworkChannel`* (and all the subclasses for cardinality).

The advantage with this approach is that the code that implements each of the channels is in one class. This would enable the implementations to be written without method calls to other classes and the associated de-referencing of *`Object`* references. This would result in a possible performance improvement.

The JCSP library does not use this approach due to the quantity of classes and the replication of code required.

3.1.2.2 Bridge Based Channels

The second approach considered when designing the implementation for the channels was to take the interface and create a new sub class for each of the required cardinalities. These classes implement the required synchronisation between the reader and the writer process. This will ensure that they are safe to use for many readers and writers as necessary.

Another interface *`ObjectStore`* is defined to define the interface to putting an *`Object`*, reading an *`Object`* and checking the state of the storage type. A class is defined which extends *`ObjectStore`* for each of the storage policies required. Provided in the library would be an implementation of *`ObjectStore`* for the following.

1. *`SingleObject`* (default if an *`ObjectStore`* if not specified)
2. *`Buffer`*
3. *`OverWritingBuffer`*
4. *`Timer`*

The channels will be created with an instance of an *`ObjectStore`* to store the data sent across the channel. The *`Channel`* classes will control the synchronisation by checking the state of the *`ObjectStore`* to find out if the reader or writer should block to wait for a writer or a reader. The `write()` method will put data into the *`ObjectStore`* and the `read()` method will get data from the ObjectStore.

The Class Diagram below shows the relationships between the classes in the bridge based model.

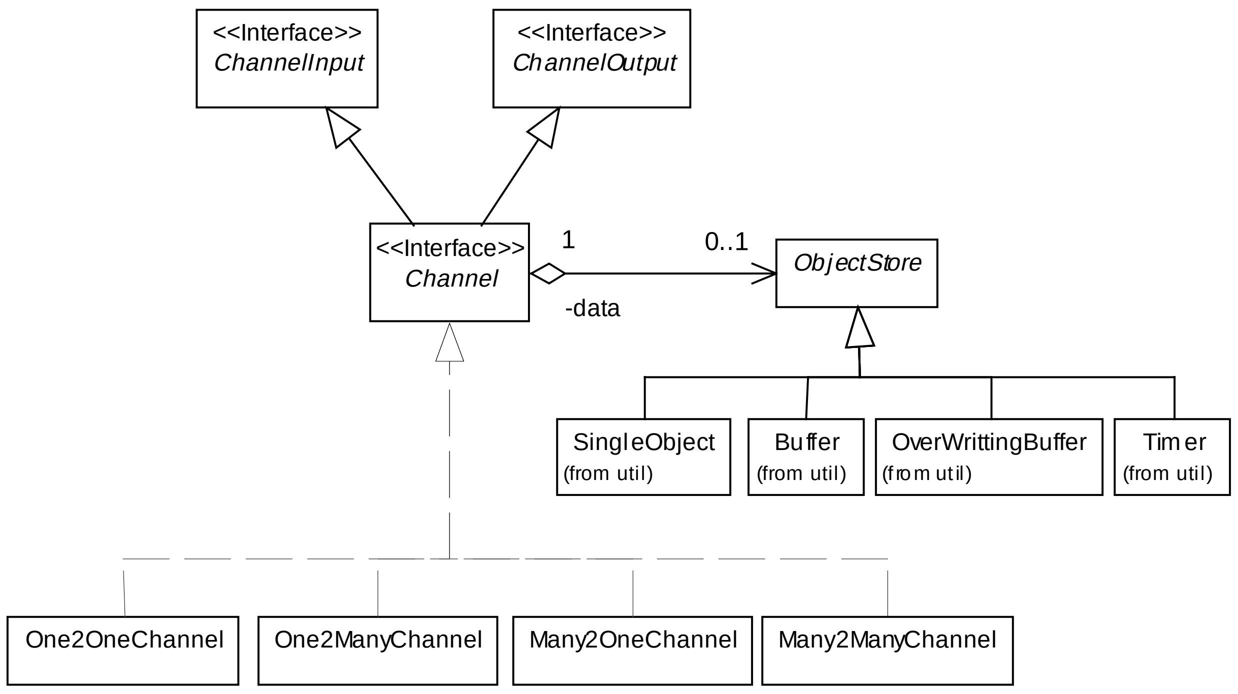


Figure 4 - Bridge Channels - Class Diagram

3.1.2.2.1 ObjectStore

The `ObjectStore` interface is an abstract class that defines several abstract protected methods that must be defined by sub-classes. The interface is shown below.

```
public abstract class ObjectStore {
  protected static final int EMPTY;
  protected static final int FULL;
  protected static final int NONEMPTYFULL;

  protected abstract void putObject(Object
o);
  protected abstract Object getObject();
  protected abstract int getState();
}
```

Code 14 - ObjectStore interface

The methods are defined as protected[3] as only the `Channel` classes defined within this package should be able to access members of the `ObjectStore`. When defining the sub-classes, the constructor of those classes must be public but all the other methods should remain protected. This gives a clean interface to the `ObjectStore` and insures data is not changed by other classes.

[3] The protected visibility in Java enables classes in the defining package as well as sub-classes access the methods of the class.

3.1.2.2.1.1 Single Object

The `SingleObject` will store the reference to the `Object` being written and a flag indicating the state of the `ObjectStore`. The state diagram below shows the allowed states and transitions of the `SingleObject` class when used in a `Channel`.

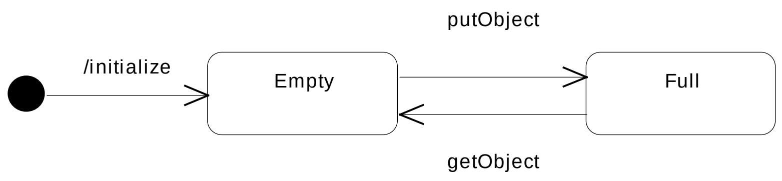


Figure 5 – SingleObject - State Diagram

3.1.2.2.1.2 Buffer

The `Buffer` will store the references to the `Objects` being written in a circular buffer, with a count of the number of items. The state will be calculated from the size of the buffer and the count. The state diagram below shows the allowed states and transitions of the `Buffer` class when used in a `Channel`.

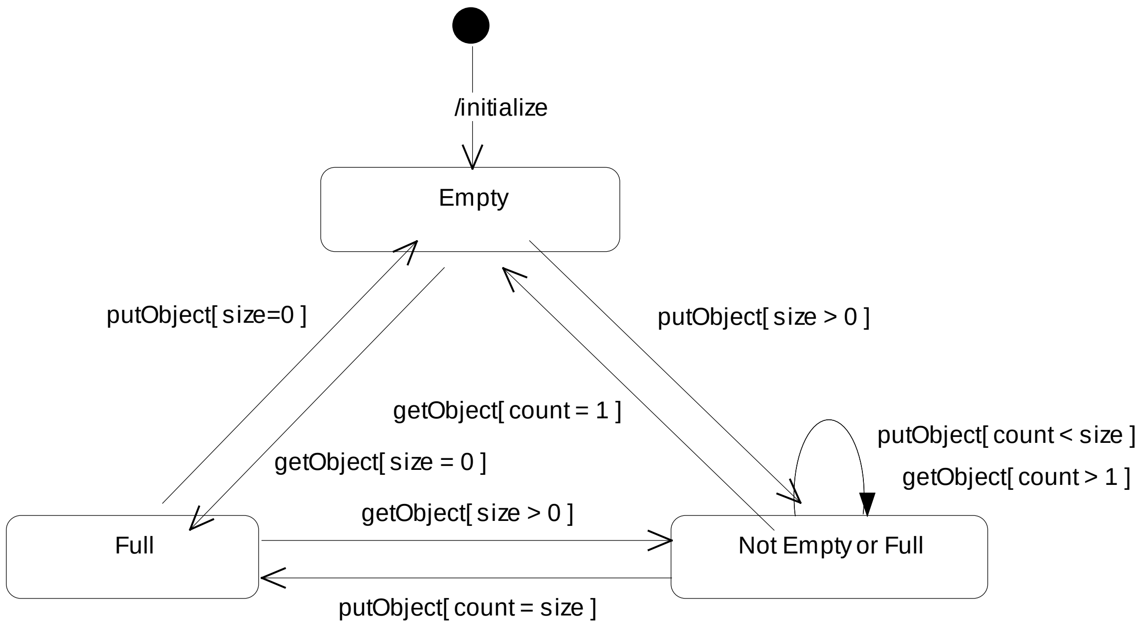


Figure 6 – Buffer - State Diagram

3.1.2.2.1.3 OverwritingBuffer

The `OverWritingBuffer` will store the references to the `Object` being written in a circular buffer, with a count of the number of items. The state will be calculated from the size of the buffer and the count, the state will either be empty if the count is 0 or NONEMPTYFULL otherwise. The state diagram below shows the allowed states and transitions of the `OverWritingBuffer` class when used in a `Channel`.

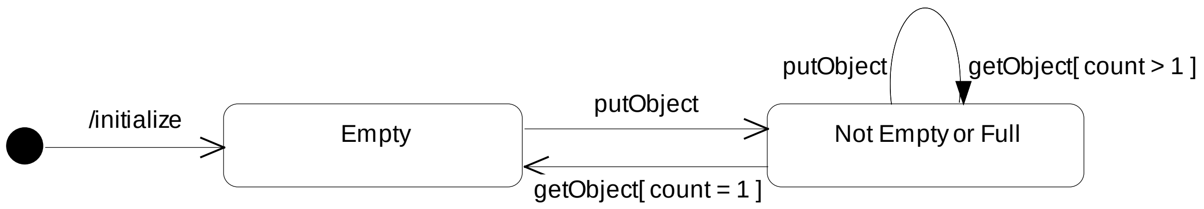


Figure 7 – OverWritingBuffer - State Diagram

3.1.2.2.1.4 Timer

The `Timer` will discard the reference to the `Object` being written. The state will always be NONEMPTYFULL. The state diagram below shows the allowed states and transitions of the `Timer` class when used in a `Channel`.

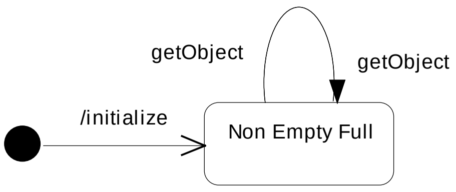


Figure 8 – Timer - State Diagram

3.1.2.2.2 The read method

The basic functionality of the `read()` method is the same for all the different channels, the channels with a cardinality of Many readers provide extra synchronisation around the method to ensure exclusive access.

The purpose of the read method on a channel is to provide the correct synchronisation for the channel between the reader and writer, and to return the next *`Object`* from the channel. The following diagram shows the interaction between the *`Channel`* and the *`ObjectStore`*.

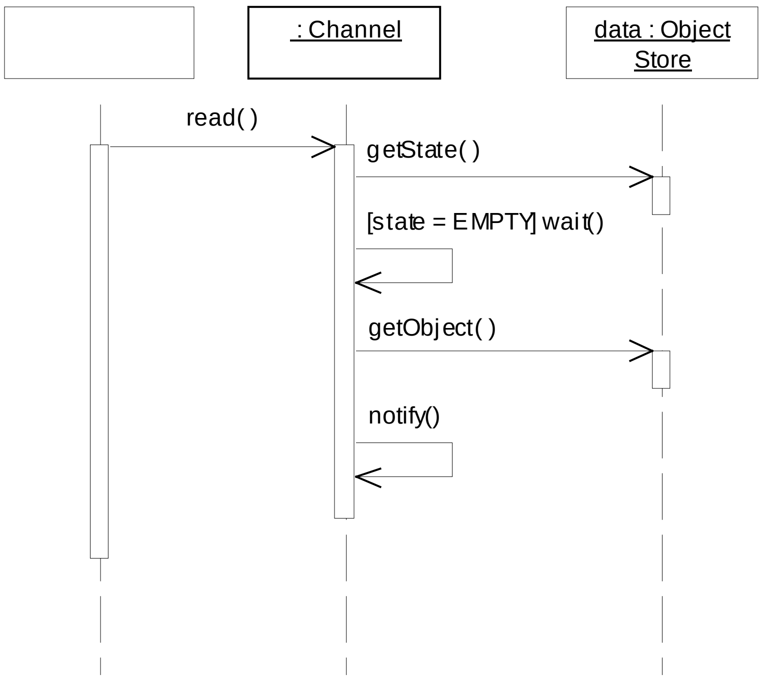


Figure 9 - Class interactions for the Channel read method

The `read()`method is a synchronised method so that only one process can access the *`Channel`*. The method will block (using the `wait()` method) if the *`ObjectStore`* is EMPTY (this provides the synchronisation between the reader and the writer). The synchronisation on the *`Channel`* monitor is released when the `wait()` method is invoked allowing the writer to call the `write()` method, the `write()` method will then notify this reader when it has finished writing the data. The reader will then regain the *`Channel`* monitor. The next *`Object`* will then be obtained using `getObject()`. The `notify()` method will then be invoked to schedule any waiting writers, the *`Object`* will then be returned.

The `read()` method for the Many reader channels will have an extra synchronisation on a read monitor object. This synchronisation is around the method described above to ensure that only one reader can be invoking `read()` at any one time. If this was not done, a second reader could come along while the first is waiting (and therefore not holding the monitor) and enter the method and may cause the *`Channel`* to be left in an inconsistent state.

NOTE: When the `wait()` call is made only the lock from synchronising on the same object as `wait()` was invoked on is released. Therefore, when the reader has synchronised on the read monitor then waited on the channel monitor no other readers will be able to gain the read monitor. The other readers will have to wait until the reader is notified and regains the channel monitor and releases the read monitor. The writers will however be able to get through the write monitor and gain the channel monitor while the reader is waiting.

3.1.2.2.3 The write method

The basic functionality of the `write()` method is the same for all the different channels. The channels with a cardinality of Many writers provide extra synchronisation around the method to ensure exclusive access.

The purpose of the `write()` method on a *`Channel`* is to provide the correct synchronisation for the *`Channel`* between the writer and reader, and to place the next *`Object`* in the *`Channel`*. The following diagram shows the interaction between the *`Channel`* and the *`ObjectStore`*.

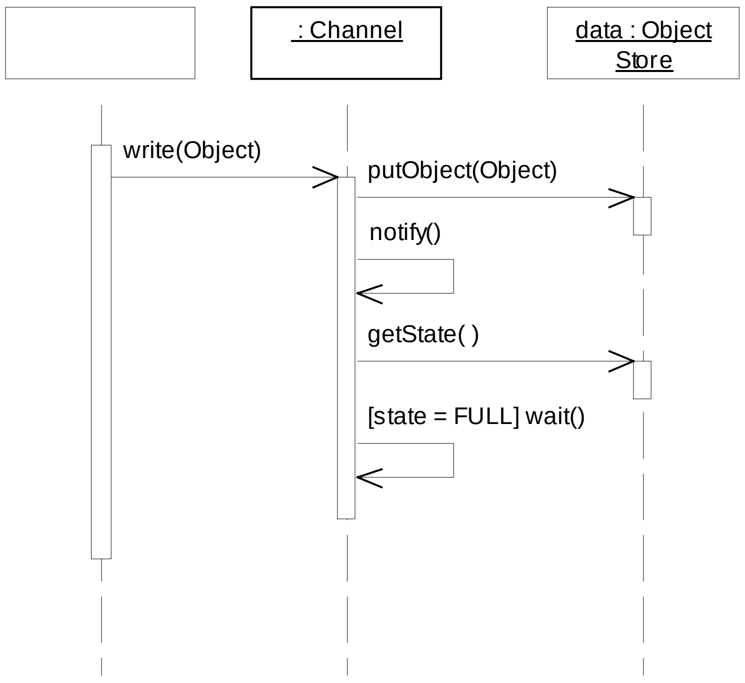


Figure 10 - Class interactions for the Channel write method

The `write()` method is a synchronised method so that only one process can access the Chan*n*el. The `putObject()` method will be invoked to store the value in the *`Channel`*. The `notify()` method will then be invoked to schedule any waiting readers. The method will block (using the wait method) if the *`ObjectStore`* is `FULL` (this provides the synchronisation between the writer and the reader). The synchronisation on the *`Channel`* monitor is released when the wait method is invoked allowing the reader to call the `read()` method, the read method will then notify this writer when it has finished reading the data. The writer will then regain the *`Channel`* monitor and exit.

The `write()` method for the Many writer channels will have an extra synchronisation on a write monitor object. This synchronisation is around the method described above to ensure that only one writer can be invoking `write()` at any one time. If this was not done, a second writer could come along while the first is waiting (and therefore not holding the monitor) and enter the method and may cause the *`Channel`* to be left in an inconsistent state.

NOTE: When the `wait()` *call is made only the lock from synchronising on the same object as* `wait()` *was invoked on is released. Therefore, when the writer has synchronised on the write monitor then waited on the channel monitor no other writers will be able to gain the write monitor. The other writers will have to wait until the writer is notified and regains the channel monitor and releases the write monitor. The readers will however be able to get through the read monitor and gain the channel monitor while the writer is waiting.*

3.2 Alternative Construct

The Alternative construct in the JCSP class library is based on the *`Alternative`* class developed by Peter Welch. The class is modified so that only *`AltingChannels`* may be selected upon and common code has been factored out into utility methods.

NOTE: The `ALT` *construct in OCCAM is actually a process. In this version of the Java* `Alternative`*, it is not a process. Future versions may be implemented differently, there was not enough time at the end of the project to describe details here.*

The addition of the *`Alternative`* construct should not radically alter the interface to existing classes.

3.2.1 Interface

3.2.1.1 Alternative

The interface to the *`Alternative`* class must provide methods to perform alternation under the following circumstances.

- Alting on input Channels only
- Alting on input Channels with SKIP
- Alting on input Channels with a timeout
- Alting on input Channels with guards
- Alting on input Channels with guards and SKIP
- Alting on input Channels with guards and a timeout

The interface will define several versions of the `select()` method with varying arguments. Each version will accept an array of *`AltingChannelInput`* (see next section for definition) containing references to the input channels. The `select()` method will return the index of the *`Channel`* in the array that is ready, or –1 if the skip or a timeout occurred.

The interface for the *`Alternative`* class is shown below.

```
public class Alternative {
  public int select(AltingChannelInput[] c);
  public int select(AltingChannelInput[] c, boolean skip);
  public int select(AltingChannelInput[] c, long msecs);
  public int select(AltingChannelInput[] c, long msecs, int nsecs);

  public int select(AltingChannelInput[] c, boolean[] guard);
  public int select(AltingChannelInput[] c, boolean[] guard,
                    boolean skip);
  public int select(AltingChannelInput[] c, boolean[] guard,
                    long msecs);
  public int select(AltingChannelInput[] c, boolean[] guard,
                    long msecs, int nsecs);
}
```

Code 15 - Alternative class - public interface

To use the *`Alternative`* interface a new *`Alternative`* is constructed. When selection is required, one of the `select()` methods is invoked with the array of *`AltingChannel`* as a parameter. The index returned from the method can either be used directly to invoke `read()` on the *`Channel`* in the array or using a switch statement as shown in the example below.

```
Alternative alt = new Alternative();
AltingChannelInput[] altChans = {in1, in2, in3};
Integer i;
switch (alt.select(altChans, 1000)) {
  case 1:
    i = in1.read();
    out.write(new Integer(1));
    out.write(i);
  break;
  case 2:
    i = in1.read();
    out.write(new Integer(2));
    out.write(i);
  break;
  case 3:
    i = in1.read();
    out.write(new Integer(3));
    out.write(i);
  break;
  default: // timeout
    System.out.println(“Time Out”);
  Break;
}
```

Code 16 – Alternative interface - example

The interface for the `Alternative` class includes facilities for `boolean` guards, timeouts and skips as parameters to the select method. Therefore, it is only necessary to modify the interface to the channels that allow alternative selection to have the required methods.

3.2.1.2 Alting Channels

To implement selection on channels there needs to be two methods. The first method will enable the selection on the channel and return true if data is immediately available. The second will disable the selection on the channel after one of the channels has been selected.

The easiest way to add alting to the existing channels it to add the declarations of the `enable()` and `disable()` methods to the `ChannelInput` class and then require the concrete channels to implement the bodies of these methods. This nieive approach has the following disadvantages.

- It is not safe to implement alting for all types of channel, the only types of channel that may implement alting on input are those which may only have one reader.

All methods defined in interfaces are declared to be public, this is would enable any class to call the `enable()` and `disable()` methods when in fact only the sub classes and the `Alternative` class should be able to call them. This requires that the methods be defined in a class in the same package as the `Alternative` class. The class should be defined as an abstarct class with the methods defined with package visibility.

- Adding the methods would require the modification of the existing interfaces, it would be better if alting could be added without modifying the existing interface.

The method used is to define two extra abstract classes. The first `AltingChannelInput` implements the `ChannelInput` interface and defines the format of the `enable()` and `disable()` methods, this class will be used to pass channels into processes which `ALT` on the channel. The second `AltingChannel` extends the `AltingChannelInput` and implments the `Channel` interface, no new methods are defined, this class is the super class of all `AltingChannels`.

The channels which enable alternative selection must extend the `AltingChannel` class and define the `enable()` and `disable()` methods instead of implementing the `Channel` interface. These are the only interface changes required. As the methods are defined to have package visibility only this package can define classes that extend `AltingChannel`.

The declaration of these new classes and the class digram is shown below.

```
public abstract class AltingChannelInput
    implements ChannelInput{
  abstract boolean enable(Alternative alt);
  abstract boolean disable();
}
public abstract class AltingChannel
    extends AltingChannelInput
    implements Channel {
}
```

Code 17 - AltingChannel - interfaces

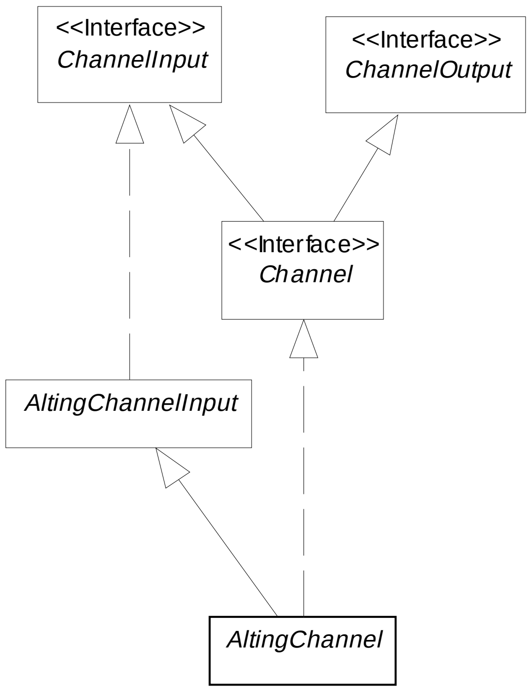


Figure 11 – JCSP Channel interface relationships with Alting

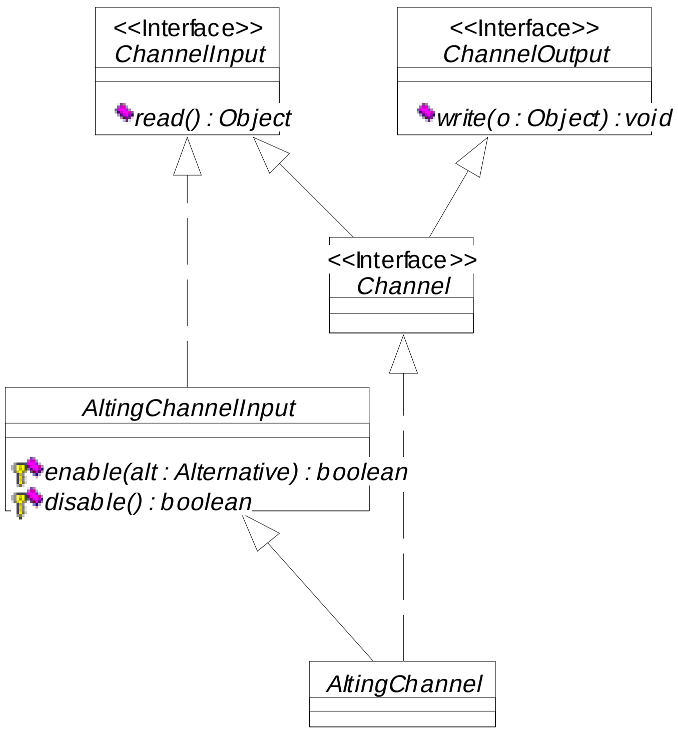


Figure 12 – JCSP Channel interfaces with Alting

3.2.2 Implementation

3.2.2.1 Alternative

The bulk of the *`Alternative`* class is mainly implemented within the `select()` methods defined in the interface with some private methods to perform some of the work. The general algorithm for selection on channels with no timeouts, guards or skips is shown below.

```
synchronize on Alternative
  for each channel c
    enable(c)
  end for
  if no Channel ready
    wait for writer
  end if
  for each channel c in reverse order
    if disable(c) = has data
      selectedChannel = index of c
    end if
  end for
  return selectedChannel
end synchronize
```

Code 18 - Algorithm for selection of Channels

To implement the selection with timeout, if no channel is ready the wait call will have a timeout value that will cause the select to awake after this time.

To implement the selection with skip, if no channel is ready there is no wait call, the channels are disabled immediately.

To implement the selection with guards the loops to enable and disable each channel will only enable/disable a channel if the `boolean` variable in the guards array for that channel is true.

The *`Alternative`* class defines a package visible method `schedule()` that must be invoked when a Channel has been enabled for selection to notify that data has become ready instead of the usual `notify()` invocation. The `schedule()` method will invoke the `notify()` method on the *`Alternative`* class.

The interaction diagram below shows the interactions between the various classes for the simple `select()` statement.

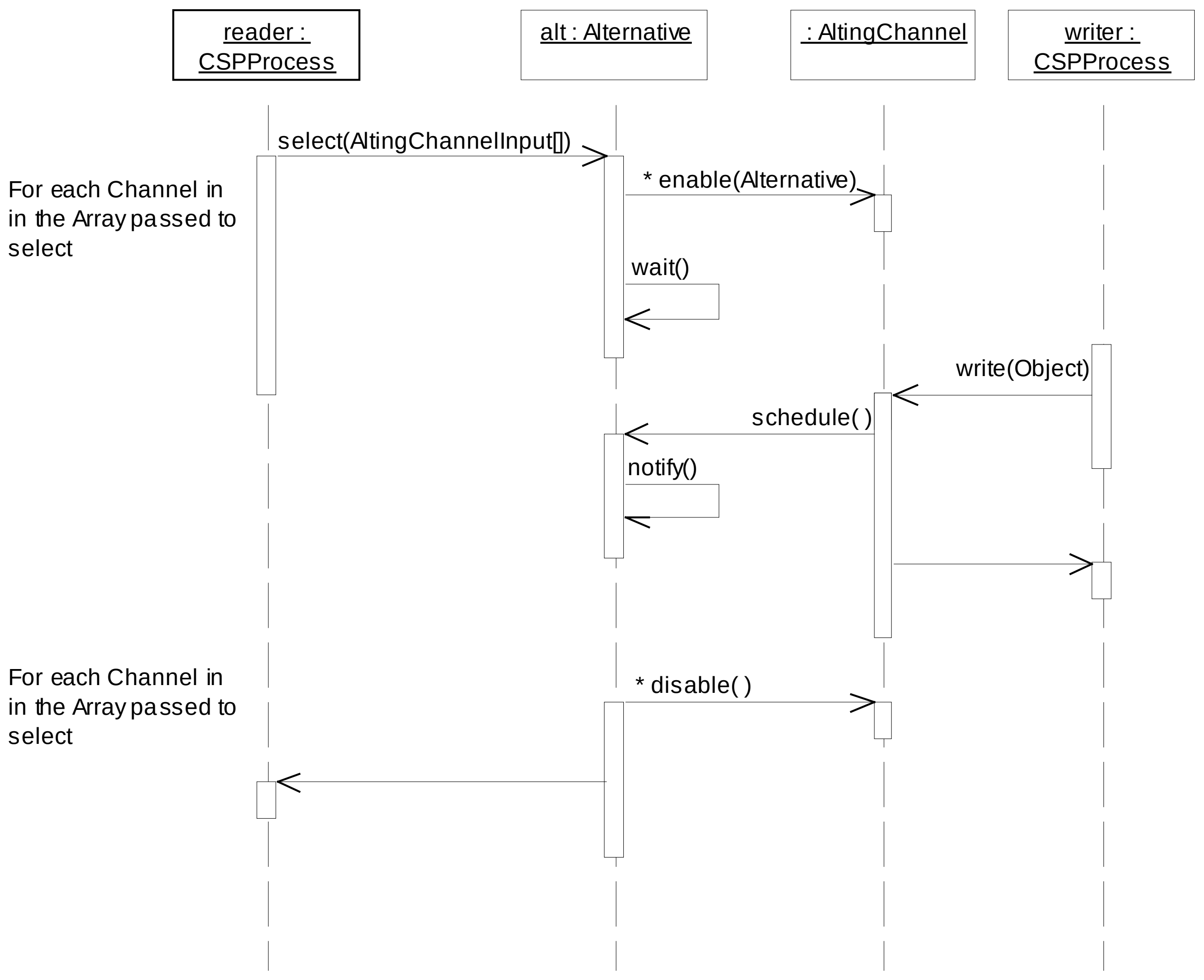


Figure 13 - Alternative - select

The enable all and disable all channels functionality is factored out into several private methods that contain the loops that perform the work. These loops have been modified so that when a channel is enabled and returns true indicating it is ready the loops will short-circuit at that point improving performance.

3.2.2.2 AltingChannel

The *`AltingChannelInput`* and *`AltingChannel`* classes have no implementation as they are abstract classes. However discussion on the implementation of the *`AltingChannel`* sub classes must be considered.

The two channels which may implement selection are *`One2OneChannel`* and *`Many2OneChannel`* as the *`Many2OneChannel`* only adds extra synchronisation on write only the implementation of the *`One2OneChannel`* needs to be considered.

To extend the *`AltingChannel`* class the implementation must define the `enable()` and `disable()` methods. The implementation of the `enable()` method simply stores the reference to the *`Alternative`* passed as the parameter in a private attribute indicating it is enabled and specifying the *`Alternative`* construct that is performing the selection. The `diable()` method sets the private attribute to `null` indicating no selection is being performed and the normal synchronisation between reader and writer should be used.

The implementation for the `write()` method must be modified so that if the private attribute for the *`Alternative`* class is not null the schedule method must be invoked on the reference otherwise `notify()` must be invoked as per normal channels.

The interaction diagram below shows the interactions involved for the version of write for *`AltingChannel`*.

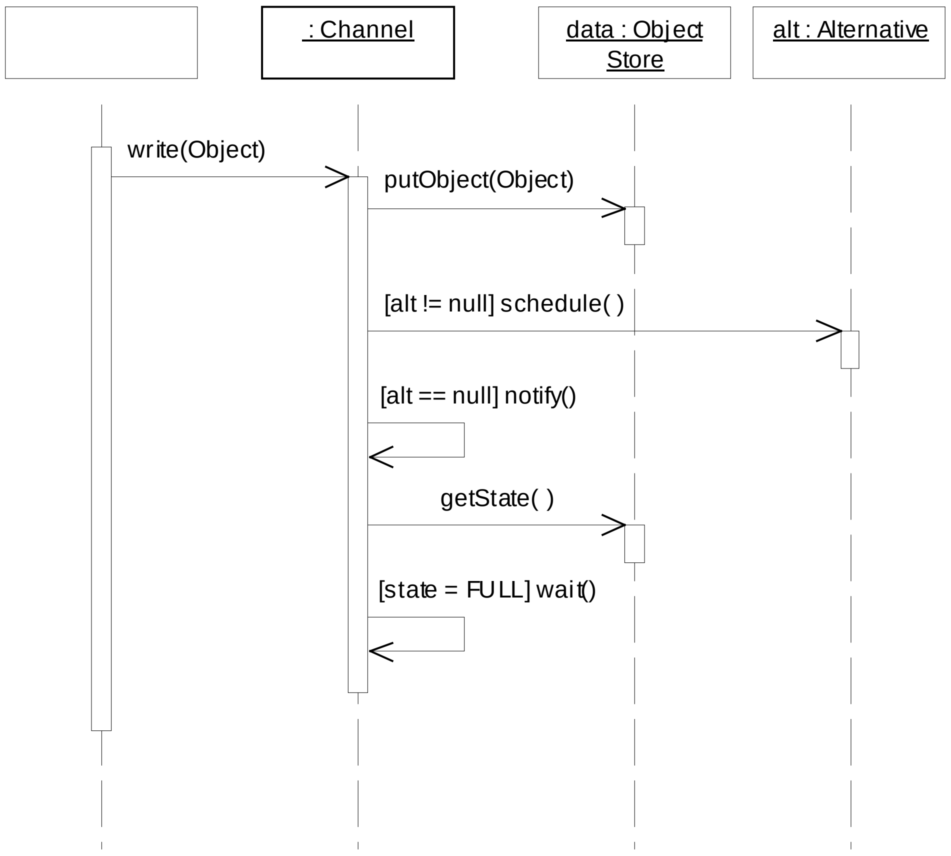


Figure 14 - AltingChannel - write

3.3 Parallel Construct and Processes

The implementation of the parallel functionality evolved through first using Java threads to encapsulate the processes with specialist versions of read and write. Followed by the development of a *`Parallel`* class that was primarily used just for reading and writing processes. Finally, the adoption and improvement of the *`Parallel`* class to model the implementation of the `PAR` construct in OCCAM with added functionality to add and remove processes.

NOTE: The current implementation does not implement `PRI PAR` *but this is being considered for future versions.*

3.3.1 Interface

3.3.1.1 Using Java Threads

The first approach to generating CSP process networks in Java was to create a new subclass of *`Thread`* for each `PROC`. The constructor of the class models the `PROC` declaration and the `run()` method models the body of the `PROC`. The constructor of the class also causes the *`Thread`* to be started when constructed. The class stores the parameters passed into the constructor in private attributes so that the `run()` method can access them.

When executing several processes in parallel the `join()` method must be invoked on each process after the last process has been constructed to wait for all of the processes to finish executing. This gives the same functionality as `PAR`. To execute a process so that the calling process is not blocked (i.e. asynchronously) the `join()` method should not be invoked.

The code below shows how a simple process is constructed and how a network of these processes is constructed.

```
import jcsp.lang.*;

class Successor extends Thread {
  private ChannelInput in;
  private ChannelOutput out;

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

  public void run() {
    while (true) {
      Integer i = in.read();
      out.write(new Integer(i.intValue() + 1));
    }
  }
}

// declarations of Delta and Prefix go here

class Nos extends Thread {
  private ChannelInput out;

  public Nos(ChannelOutput out) {
    this.out = out;
    start();
  }

  public void run() {
    Channel a = new One2OneChannel();
    Channel b = new One2OneChannel();
    Channel c = new One2OneChannel();
    Prefix p = new Prefix(new Integer(0), a, b);
    Delta d = new Delta(b, out, c);
    Successor s = new Successor(c, a);

    // wait for the processes to finish
    p.join();
    d.join();
    s.join();
  }
}
```

Code 19 – Parallelism using threads

3.3.1.2 OCCAM style PAR Construct

After using the first method to develop some test and demonstration programs, it became apparent that there were several problems with this approach.

- Definition of a process was long
- Construction of process networks was long winded
- Long winded approach to waiting for all of the processes to terminate
- Execution of fine grained processes such as channel reads and write still required parallel versions or a `PAR` construct.
- Difficult to control the execution of all the processes in the `PAR` construct
- Processes could not be re-used after termination

To overcome the problems a new approach was needed one that followed the OCCAM/CSP model. This section describes the interfaces to the classes required to implement the `PAR` construct in Java as similar to the OCCAM version as possible. There are three classes

- *`CSProcess`* (`PROC`)
- *`Parallel`* (`PAR`)
- *`ProcessNetwork`* (asynchronous execution of a *`CSProcess`*)

3.3.1.2.1 CSProcess

The *`CSProcess`* interface defines the methods that all processes that are to be executed using the Parallel construct must implement. This is equivalent to defining named processes (`PROCs`) in OCCAM.

Unlike the OCCAM PAR construct, it is not possible to execute single statements in parallel, only classes that implement the *`CSProcess`* interface can be executed in parallel.

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

Code 20 – CSProcess interface

The `run()` method of classes which implement the *`CSProcess`* interface will contain the body of the process.

To pass channels and parameters to the process the classes that implement the *`CSProcess`* interface the class must be defined with a constructor that accepts the parameters and store references to the parameters in private attributes. To execute the *`CSProcess`* it can either be run as part of the *`Parallel`* Construct (equivalent to `PAR`), *`Sequential`* construct (equivalent to `SEQ`) or the `run()` method can be invoked (equivalent to `SEQ`). The following Code demonstrates how to create a simple process.

```
public class ProcessExample implements CSProcess {
  private Object o;
  private ChannelOutput out;

  public ProcessExample(Object o, ChannelOutput out)
{
    this.o = o;
    this.out = out;
  }

  public void run() {
    while (true) {
      out.write(o);
    }
  }
}
```

Code 21 – CSProcess example

3.3.1.2.2 Parallel

The design of the `Parallel` class aims to provide similar functionality as the PAR construct in OCCAM that is also easy to use. The interface has constructors that create a new PAR construct with an array of *`CSProcess's`* or with no processes. The `Parallel` class implements the *`CSProcess`* and defines the `run()` method to execute each of the *`CSProcess's`* once in `Parallel` and returns only when all the *`CSProcess's`* finish executing.

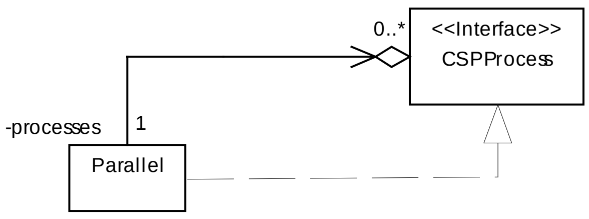


Figure 15 - Relationship between Parallel and CSProcess

The interface also defines methods to add and remove *`CSProcess's`* from the `Parallel` class. There are two `addProcess(…)` methods one that will add a single *`CSProcess`* and another that will add and array of *`CSProcess's`*. The `removeProcess(…)` method will remove the *`CSProcess`*. These methods are synchronized (as is the `run()` method) which means processes can only be added or removed if the `run()` method (or any of the other methods) is currently being executed.

```
public class Parallel implements CSProcess {
  public Parallel();
  public Parallel(CSProcess[] process);
  public synchronized void addProcess(CSProcess process);
  public synchronized void addProcess(CSProcess[] processes);
  public synchronized void removeProcess(CSProcess process);
  public synchronized void run();
}
```

Code 22 – Parallel interface

NOTE: The `Sequential` *class has the same interface as the* `Parallel` *class.*

The code below shows how a simple process is constructed and how a network of these processes is constructed.

NOTE: The example uses an advanced feature of the Java language to declare an array of `CSProcess's` *as a parameter to the* `Parallel` *constructor. This makes the code for using* `PAR` *look similar to the OCCAM version.*

```
import jcsp.lang.*;

class Successor implements CSProcess {
  private ChannelInput in;
  private ChannelOutput out;

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

  public void run() {
    while (true) {
      Integer i = in.read();
      out.write(new Integer(i.intValue() + 1));
    }
  }
}

// declarations of Delta and Prefix go here

class Nos implements CSProcess {
  private ChannelInput out;

  public Nos(ChannelOutput out) {
    this.out = out;
  }

  public void run() {
    Channel a = new One2OneChannel();
    Channel b = new One2OneChannel();
    Channel c = new One2OneChannel();
    new Parallel(new CSProcess[] { // PAR
      new Prefix(new Integer(0), a, b),
      new Delta(b, out, c),
      new Successor(c, a)
    }).run(); // run the processes and wait for them to
terminate
  }
}
```

Code 23 – Parallel Example

3.3.1.2.3 ProcessNetwork

In the Java version it would be useful if a process network could be constructed and be executed asynchronously and to have a mechanism to suspend, resume or stop the entire network when it is no longer required.

The simplest method of performing this functionality would be to create a new `Thread` within a new `ThreadGroup` to execute the `CSProcess`. The problem with this method is it requires the software constructor to perform several steps to implement the functionality. In addition, if the class library were re-written to implement some new form of parallelism all other code would have to be re-written to support the new interface.

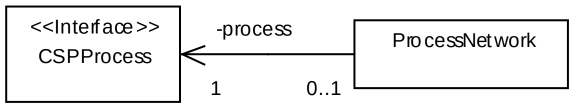


Figure 16 - Relationship between ProcessNetwork and CSProcess

The interface of the class is shown below.

```
public class ProcessNetwork {
  public ProcessNetwork(CSProcess process);
  public void stop();
  public void start();
  public void suspend();
  public void resume();
  public void join();
}
```

Code 24 – ProcessNetwork interface

3.3.2 Implementation

3.3.2.1 Using Java Threads

There is no implementation required to implement the *`Parallel`* class using just Java Threads, as there are no extra classes to be implemented. The implementation would consist of a set of guidelines on how to construct processes and networks. As this method is not going to be used, the guidelines have not been written.

3.3.2.2 OCCAM style PAR Construct

The implementation of the *`Parallel`* construct described below uses the Java Thread mechanisms to provide the parallel execution of processes. As the external interfaces to the classes have been designed to hide the actual implementation it would be possible to change the actual implementation of the *`Parallel`* class to provide some other more efficient mechanism in the future.

There are four classes used to implement the parallel functionality, the interfaces for *`CSProcess`*, *`Parallel`* and *`ProcessNetwork`* were defined earlier. The *`ParThread`* class is used in the implementation of *`Parallel`* to execute the processes in parallel. Figure 17 shows the relationships between the classes.

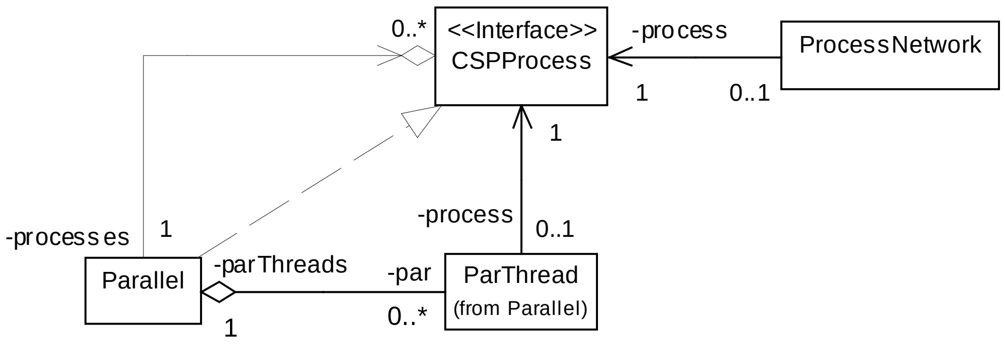


Figure 17 - Parallel Class Diagram

The *ProcessNetwork* executes one *CSProcess*.

The *Parallel* class implements the *CSProcess* interface, has zero or more *CSProcess* classes to be executed in parallel and has zero or more *ParThreads* which execute each of the *CSProcess's*.

3.3.2.2.1 CSProcess

As the *CSProcess* is an interface, there is no implementation to be described.

3.3.2.2.2 Parallel

The *Parallel* class executes each of the *CSProcess's* once in parallel when the `run()` method is called, any threads that are created are done within the `run()` method. The pseudo code below shows the basic functionality of the run method.

```
for each process
  restart process
end for
wait for processes to finsih
```

Code 25 – Parallel run() method – pseudo code

The simplest method of implementing the `run()` method would be to create a special *Thread* class that would call a method on the *Parallel* class before the process is executed to increment a reference count. The *Thread* would then execute the `run()` method of the *CSProcess*. When the *CSProcess* terminates the *Thread* would then execute another method on the *Parallel* class that would decrement the reference count, if this was the last process the method would then notify the run method and wake it up. The *Thread* would then terminate. This simple approach would require a new set of threads to be created for each call to the `run()` method causing extra overhead.

The approach used is similar to described as above except the special *Thread* (*ParThread* class) is actually a class that implements the *Runnable* interface. The constructor of this class creates a new *Thread* to execute the `run()` method and starts it running. The `run()` method of the

ParThread has an infinite loop that executes `wait()` at the beginning of the loop, then executes the *CSProcess* and then executes the method on the *Parallel* class to decrement the reference count. As the *Thread* now does not terminate it can be used to execute the *CSProcess* more than once thus saving on the overhead of *Thread* creation (apart from at initialisation).

The *ParThread* class defines a `start()` method which is called by the *Parallel* class to start the *CSProcess* running. The `start()` method calls the method on the *Parallel* class to increment the reference count and then calls notify to wake up the `run()` method on the *ParThread*.

The interaction diagram shows the sequence of method calls involved in the *Parallel* class's `run()` method. The diagram includes some extra method names (`enter()` and `finish()`) which have until now not been named.

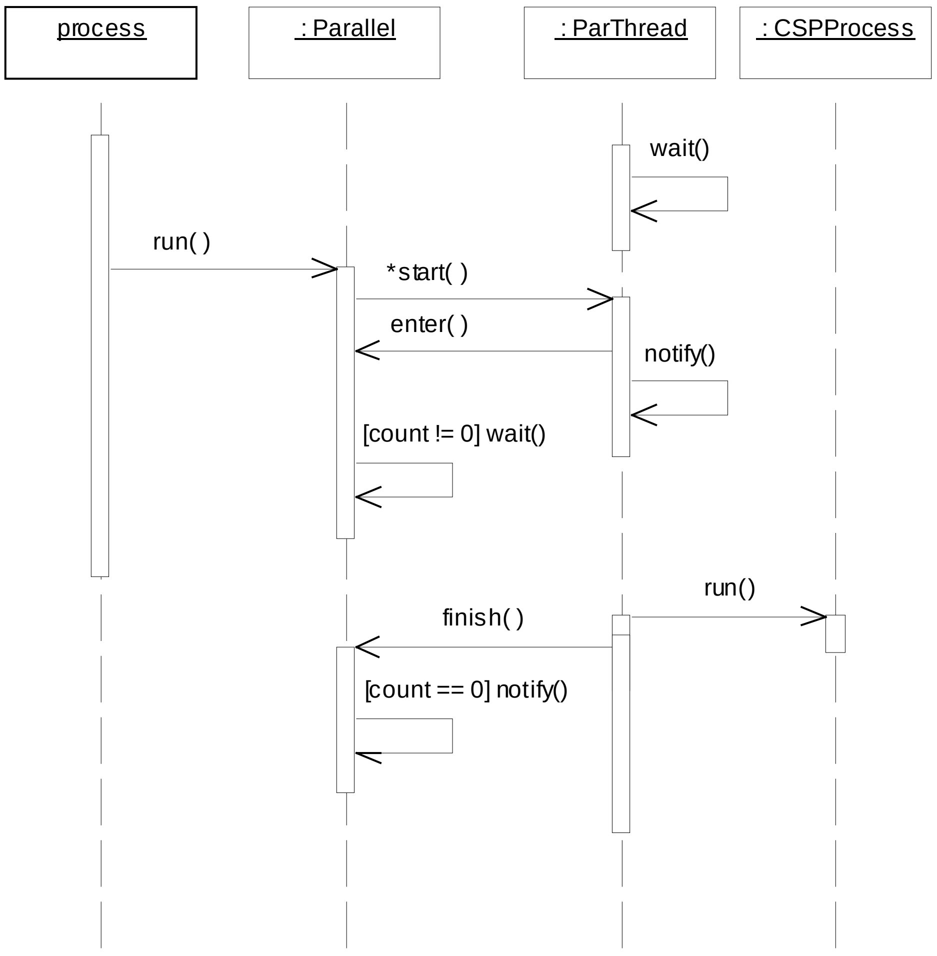


Figure 18 - Parallel – run()

There is one last point that need to be considered in the design of the *ParThread* class. This is the topic of race hazards. It would be possible for the constructor to return before the `run()` method has been started by the *Thread* (and therefor the wait() method not reached). The *Parallel* class could then call the start() method on the *ParThread* (causing notify() to be invoked). The *ParThread* would then be waiting for a notification that has already occurred.

This race hazard can be avoided by including a synchronised block around the start() method invocation on the *Thread* in the constructor. In this synchronised block would also be a wait() invocation to wait for the thread to be created and the run() method invoked. At the top of the run method there would be a notify() to indicate the thread has started. Once the wait() call is reached in the run() method the control would then return to the constructor. The race hazard is now removed.

The enter() method is called by the *ParThread* class to increment the reference count of the number of *CSProcess's* being executed.

The finish() method is called by the *ParThread* class to decrement the reference count of the number of *CSProcess's* being executed. If the count is 0 the run() method will be notified.

The *Parallel* class will store the *CSProcess's* to be executed in an array, when the class is constructed the array will be created and the references to the *CSProcess's* copied into the new array. There is a flag processesChanged, which will be set to true if the number of *CSProcess's* into the array changes.

The addProcess() method will add a *CSProcess* to the array by increasing the size of the array and copying the old array into the new one, the processesChanged flag will be set to true.

The removeProcess() method will search through the array to find the CSProcess. The old array will be copied into a new one apart from the removed *CSProcess*, the processesChanged flag will be set to true.

The *ParThreads* created by the *Parallel* class will be stored in an array and not disposed of when processes are removed. The *ParThread's* will be created by the run() method so that the *ProcessNetwork* class functions correctly.

When the run() method is invoked the processesChanged flag is checked. If it is false a loop will invoke start() on each of the *ParThreads* to set them running again. If it is true a first loop will loop through the *ParThread* array for each of the *CSProcess's*, the setProcess() method will be invoked to set the *CSProcess* for that *ParThread* to the *CSProcess* and then it will be started. A check will be made to see if the *CSProcess* array is greater than the *ParThread* array, if this is the case new *ParThreads* must be created for each of the new *CSProcess's*. The *ParThread* array will be enlarged to store the new *ParThreads*. A loop will then create a new *ParThread* and the start() method will then be invoked. The processesChanged flag will be set to false.

As described above *ParThreads* are created as they are needed by the run() method and only done so when there are less *ParThreads* than *CSProcess's*.

If a program was written using a *Parallel* construct the program should terminate when the run() method returns. The implementation so far does not do this. The reason is that the Java program does not terminate until all non-daemon Threads have terminated. In the implementation of *Parallel* there are non-terminating threads. To ensure the program terminates the threads create by *ParThread* should have their daemon flag set to true.

The following state diagram shows the possible states and actions possible in the *Parallel* class.

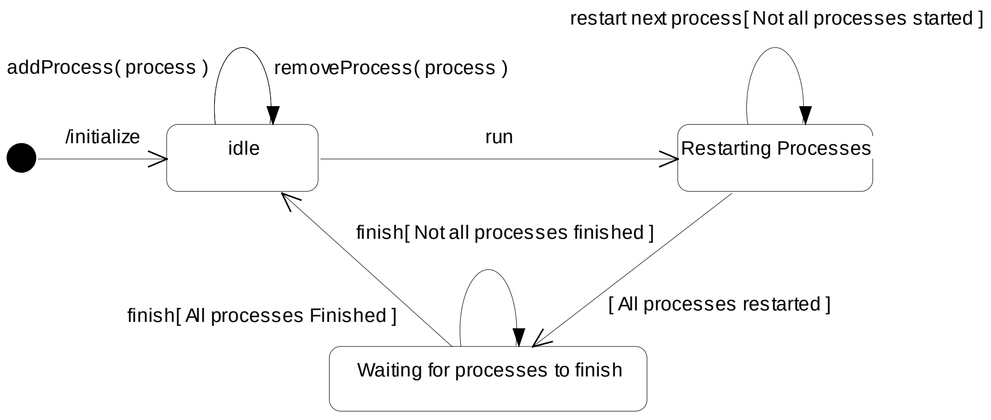

restart next process[Not all processes started]
addProcess(process)
removeProcess(process)
/initialize
idle
run
Restarting Processes
finish[Not all processes finished]
finish[All processes Finished]
[All processes restarted]
Waiting for processes to finish

Figure 19 - Parallel State Diagram

One final performance improvement can be introduced that reduces the number of *ParThreads* created. In the run method only n-1 *ParThreads* are created (where n is the number of *CSProcess*). The last *CSProcess* will be executed in the same *Thread* as the run() method by the run() method being invoked on the *CSProcess* method the wait() method is called.

3.3.2.2.3 ProcessNetwork

The *ProcessNetwork* class is implemented as a *Runnable* class that executes the run() method of the *CSProcess* which is to be executed asynchronously. The class creates a new *ThreadGroup* that will contain the *Threads* of all the *CSProcess's* created by the top *CSProcess* or any sub processes. The *ThreadGroup* will be used to control the execution of the entire network. A Thread will be created in the *ThreadGroup* to execute the run() method of this class.

The start() method will invoke start() on the *Thread*, this will cause the threads for the network to be started.

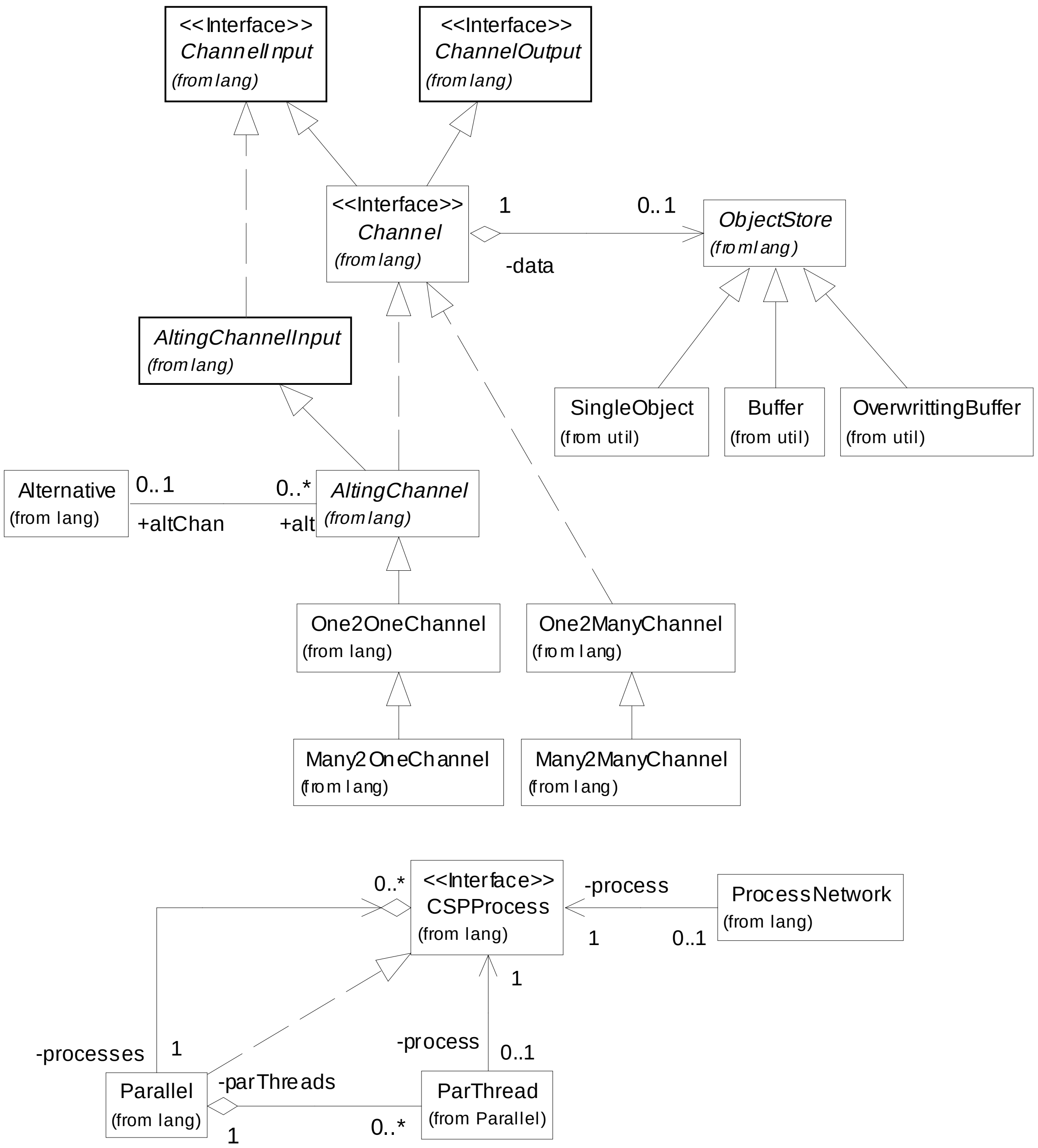
The stop(), suspend(), resume() methods will invoke the corresponding method on the *ThreadGroup* which will invoke the corresponding method on each *Thread*.

The join() method will invoke join() on the *Thread*, this will cause the current *Thread* to wait for the thread to stop executing.

Appendices

Appendix A Class Diagrams

A.1 JCSP Language Classes Overview



<<Interface>> ChannelInput (from lang)
<<Interface>> ChannelOutput (from lang)
<<Interface>> Channel (from lang)
1
0..1
-data
ObjectStore (from lang)
AltingChannelInput (from lang)
SingleObject (from util)
Buffer (from util)
OverwrittingBuffer (from util)
Alternative (from lang)
0..1
+altChan
0..*
+alt
AltingChannel (from lang)
One2OneChannel (from lang)
One2ManyChannel (from lang)
Many2OneChannel (from lang)
Many2ManyChannel (from lang)
0..*
<<Interface>> CSPProcess (from lang)
-process
1
0..1
ProcessNetwork (from lang)
1
-processes
1
-process
0..1
Parallel (from lang)
-parThreads
1
0..*
ParThread (from Parallel)

Figure 20 - JCSP Language classes – Overview

A.2 JCSP Language Classes Detail

A.2.1 Channel Classes

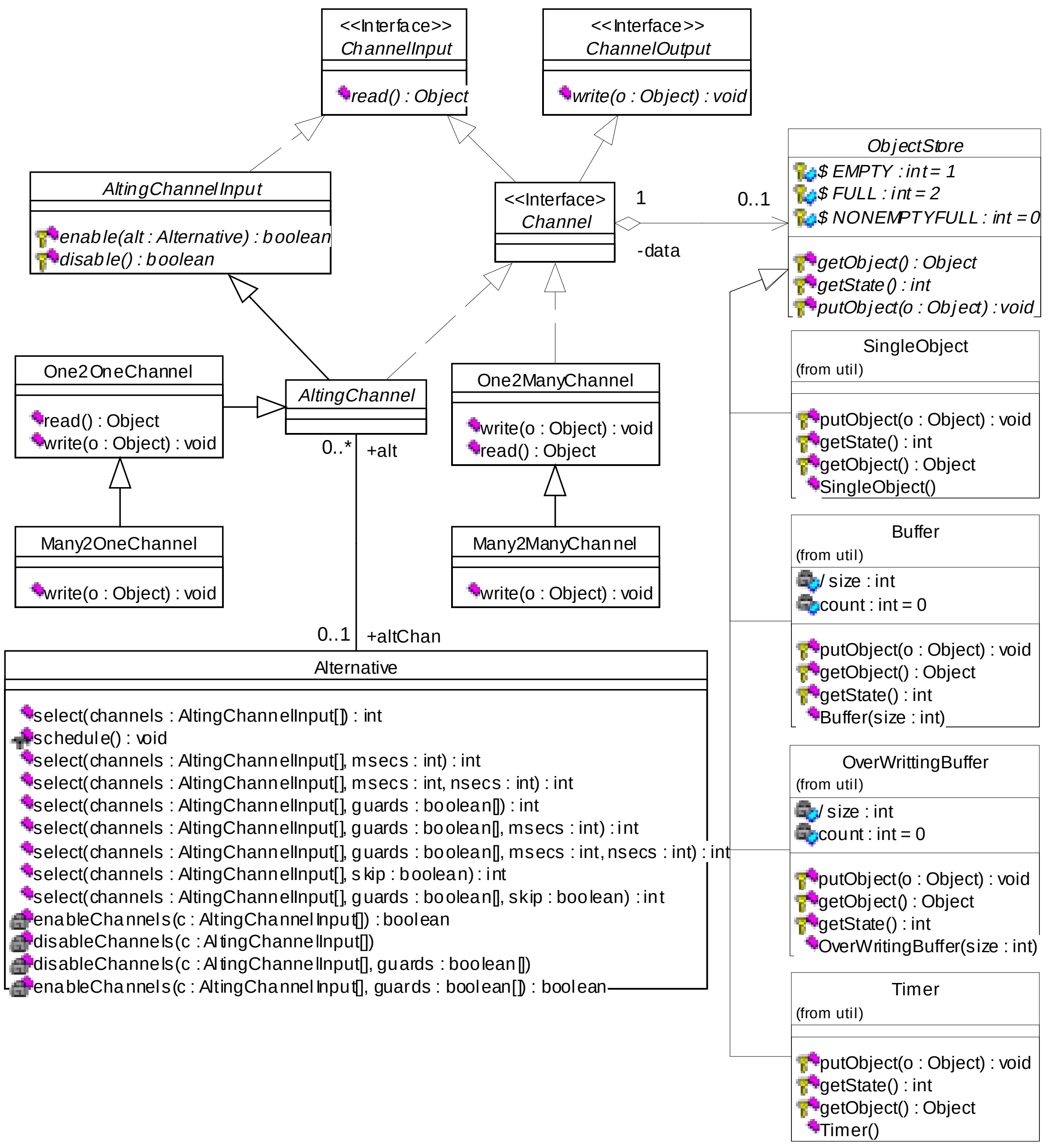

<<Interface>>
ChannelInput
read() : Object
<<Interface>>
ChannelOutput
write(o : Object) : void
ObjectStore
$ EMPTY : int = 1
$ FULL : int = 2
$ NONEMPTYFULL : int = 0
getObject() : Object
getState() : int
putObject(o : Object) : void
AltingChannelInput
enable(alt : Alternative) : boolean
disable() : boolean
<<Interface>>
Channel
1
0..1
-data
SingleObject
(from util)
putObject(o : Object) : void
getState() : int
getObject() : Object
SingleObject()
One2OneChannel
read() : Object
write(o : Object) : void
AltingChannel
One2ManyChannel
write(o : Object) : void
read() : Object
0..*
+alt
Buffer
(from util)
/ size : int
count : int = 0
putObject(o : Object) : void
getObject() : Object
getState() : int
Buffer(size : int)
Many2OneChannel
write(o : Object) : void
Many2ManyChannel
write(o : Object) : void
0..1
+altChan
Alternative
select(channels : AltingChannelInput[]) : int
schedule() : void
select(channels : AltingChannelInput[], msecs : int) : int
select(channels : AltingChannelInput[], msecs : int, nsecs : int) : int
select(channels : AltingChannelInput[], guards : boolean[]) : int
select(channels : AltingChannelInput[], guards : boolean[], msecs : int) : int
select(channels : AltingChannelInput[], guards : boolean[], msecs : int, nsecs : int) : int
select(channels : AltingChannelInput[], skip : boolean) : int
select(channels : AltingChannelInput[], guards : boolean[], skip : boolean) : int
enableChannels(c : AltingChannelInput[]) : boolean
disableChannels(c : AltingChannelInput[])
disableChannels(c : AltingChannelInput[], guards : boolean[])
enableChannels(c : AltingChannelInput[], guards : boolean[]) : boolean
OverWrittingBuffer
(from util)
/ size : int
count : int = 0
putObject(o : Object) : void
getObject() : Object
getState() : int
OverWritingBuffer(size : int)
Timer
(from util)
putObject(o : Object) : void
getState() : int
getObject() : Object
Timer()

Figure 21 - JCSP Channel classes - Detail

A.2.2 Parallel Classes

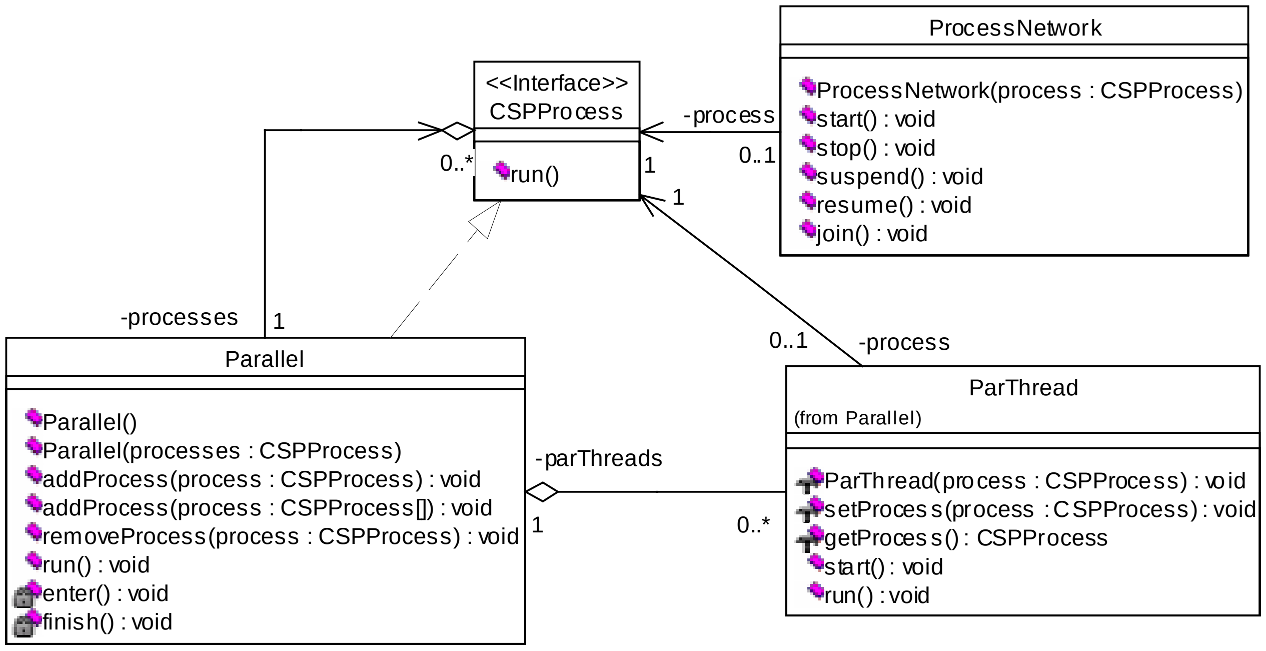

ProcessNetwork
ProcessNetwork(process : CSPProcess)
start() : void
stop() : void
suspend() : void
resume() : void
join() : void
<<Interface>>
CSPProcess
run()
-process
0..1
1
0..*
1
-processes
1
Parallel
Parallel()
Parallel(processes : CSPProcess)
addProcess(process : CSPProcess) : void
addProcess(process : CSPProcess[]) : void
removeProcess(process : CSPProcess) : void
run() : void
enter() : void
finish() : void
0..1
-process
ParThread
(from Parallel)
ParThread(process : CSPProcess) : void
setProcess(process : CSPProcess) : void
getProcess() : CSPProcess
start() : void
run() : void
-parThreads
1
0..*

Figure 22 - JCSP Parallel classes - Detail

Bibliography

[1] Martin Fowler with Kendall Scott. 1997, *'UML Distilled'*, Addison Wesley, ISBN 0-201-32563-2

[2] CGS-THOMSON Microelectronics Ltd. 1995, *OCCAM 2.1 'reference manual'*, Prentice Hall.

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

[4] JavaSoft. 1997, *'JDK™ 1.1 Documentation'*, Sun Microsystems. `http://www.javasoft.com/products/jdk/1.1/docs/`

[5] Peter .H. Welch. 1996, *'Java Channels: A learning Experience'*, University of Kent Canterbury, England.

[6] Peter .H. Welch. 1998, *'Java Threads in the Light of OCCAM/CSP' p259-284 in 'WoTUG-21: Architectures, Languages and Patterns for Parallel and Distributed Applications'*, IOS Press (Amsterdam), ISBN 9051993919.

[7] Gerald H. Hilderink. 1997, *'Communicating Java Threads Reference Manual' p283-325 in 'WoTUG-20: Parallel Programming and Java'*, IOS Press (Amsterdam), ISBN 9051993366.

[8] Gamma. Helm, Johnson, Vissides. 1995, *Design Patterns (Elements of Reusable Object-Oriented Software)*, Addison Wesley, ISBN 0-201-63361-2