The Mjølner BETA System
Process Library
Reference Manual

Mjølner Informatics Report

MIA 94-29(1.0)

September 1995

Contents

1 Introduction

This document describes the version 1.4 of the process library in the Mjølner BETA System. This library implements support for manipulating operating system processes and for communicating with them.

`Processmanager` and OSInterface

The fragments dealing with the manipulation of processes are `processmanager` and `osinterface`. `Processmanager` supports starting a child process, stopping it, and similar things. `Osinterface` supports getting information about the run-time environment of the process itself, such as the name of the host on which it runs.

Communication and SystemComm

The fragments dealing with communication between processes are `communication` and `systemComm`. They are very much alike the interface, but they are constructed to run in different environments:

`SystemComm` demands that the program uses the BETA simulated concurrency, i.e. the slot `program:descriptor` must be a specialization of `systemenv`. In return, one does not have to explicitly transfer the thread of control by `suspending` when a `systemComm` operation is about to block - the `systemenv` scheduler and `systemComm` cooperate to make it look like implicit scheduling. This ensures that co-routines which can proceed with their work will never be prevented from this because of a blocking communication operation in some other co-routine.

`Communication` will work independently of `systemenv`, but here any communication which cannot be carried out at once will block the program by default. There are notification hooks, which make it possible for the programmer to cancel operations which are about to block, and other hooks (`idle` hooks) which make it possible to keep the program alive while a lengthy operation is proceeding.

`Communication` is simpler to use for basic tasks, but for more complicated tasks, `systemComm` provides a considerably stronger and more flexible base.

Some aspects of support for the communication between processes have been separated into the fragments `commError`, `commAddress` and `errorCallback`. `CommError` simply defines a number of constants. `CommAddress` defines a hierarchy of patterns, which model addresses (destinations for communications) in a platform independent way. `ErrorCallback` defines a few patterns used for error handling in this library.

Connections

On top of the support for single communication connections, `connectionPool` implements support for holding a set of connections, and providing concurrency-secure access to these connections by means of platform independent addresses, i.e. instances of patterns in `commAddress`. This abstracts away the need to open and close these connections: if connections to the required destination is available, one of them will be used, otherwise a new connection will automatically be opened. If the process hits a maximum limit for the number of open connections, a least recently used (and currently unused) connection will be closed.

Extending `systemenv`

The last part of the process library extends the `systemenv` framework for concurrency within one BETA process. `IdScheduler` implements support for sub-contracting the job of the scheduler: Client co-routines can suspend themselves, identifying what they are waiting for by means of an integer `id`; later, a managing co-routine can wake up clients selectively, using such `id`s.

2 Manipulating Processes

First, a bit of terminology. A binary file is a diskfile, from which the operating system is able to create a process, which is then called an instance of the binary. A process is a dynamic entity within a computer which has an internal state and may interact with other processes. So there may be more than one process which is instantiated from any given binary file, and these processes are by no means the same thing. Here, each BETA object which is an instance of the pattern `process`, models one process. If you want to manipulate more than one instantiation of a given binary, use more than one `process` object.

2.1 Child Processes

The fragment `processmanager` is concerned with child processes. An instance of the `process` pattern in this fragment is attached to a binary file by initializing it with a file specification, like

```
'/bin/someApplication' -> aProcess.init;
```

Process

In the following, `aProcess` denotes an instance of the pattern `process`, which has been attached to a binary file.

Arguments and instantiating

One has the option to set up arguments for an instantiation of the binary, using `aProcess.argument.append`, once for each argument. Afterwards, the process can be instantiated with `aProcess.start`. In the following, this instantiation is referred to as the child process. When it has been started, it is possible to change its life cycle and to adjust to it: `aProcess.stop` causes the child process to be killed, `aProcess.awaitStopped` causes this process to sleep until the child process terminates, and `aProcess.stillRunning` is a predicate which returns true if the child process has not yet terminated.

The `onStart` virtual is a hook, into which one can put code to be executed immediately after the child process has been started, and the `onStop` virtual is a hook which is executed when `stop` has stopped the process. Please notice that `onStop` will NOT be executed in the (typical) case when the child process terminates for any other reason, e.g. when it terminates normally.

Inter-process communication

The remaining pattern attributes of `process` are concerned with inter-process communication. The network of inter-process communication must be defined before the child processes are started. `ConnectToProcess` and `connectInPipe` enter a reference to another `process` object and connect the referred child processes in a pipeline. `redirectFromFile` arranges for the child process to take standard input from the specified file, and `redirectToFile` makes it redirect standard output to the given file.

Finally, `redirectFromChannel` enters the `writeEnd` of a `pipe` and makes the child process accept standard input from that pipe, and `redirectToChannel` enters the `readEnd` of a `pipe` and makes the child process send standard output to it. The entered parameter is declared to be a (specialization of a) stream. The reason for this is that a future release may accept a broader range of types of objects entered; it should, for instance, be possible to use sockets.

2.2 This Process and its Environment

The fragment `osinterface` contains the pattern `osinterface`, which supports access to the run-time environment of this process. To use it, create an instance and initialize it with `init`.

Then `hostMachine` will return a text characterizing the combination of the type of machine and operating system on which this process runs, such as “sun4s” on a Sun SparcStation Classic running SunOS 5.3 (Solaris). This is the same as the name of the architecture dependent directories in the BETA directory hierarchy. `HostName` returns the name of this host; `getHostAddr` returns the internet address of this host, in a format like “130.225.16.15”. Finally, `thisProcess` is an instance of `process` referring to this process. It is kept around for backwards compatibility but otherwise obsolete: Scanning the command line arguments to this process is now supported in `betaenv`, and the other operations are not relevant on this process, as they must be executed before the process is actually instantiated.

3 Communicating with other Processes

Two quite similar libraries are available for exploiting inter-process communication. This section presents the basic concepts, which apply to both libraries. Two subsequent sections describe them in greater detail. At that level, differences exist.

3.1 Communication Concepts

Inter-process communication is usually described as “message based” or as “connection based”. In both cases, any primitive communication act has a number of participants, playing roles as the receiving or the transmitting end. In this context, there will always be exactly one transmitting party and one receiving party. There is support for specifying a group address, but there is not currently any ready-made implementation of a group communication protocol.

For a message based communication, each message is sent to an explicitly specified receiver. For a connection based communication, at first a connection between two parties is established. From that point, messages can be transmitted via this connection without any explicit reference to their destination. Here, the model of communication is connection oriented.

Pipes

For operating systems that support a notion of standard channels for receiving input and delivering output and possibly other things, it is possible for the communicating processes to be unaware (i.e. independent) of the fact that standard input comes from another process or that standard output goes to another process: It all looks the same as if the data came from a keyboard and went to a display or whatever. On the other hand, this level of abstraction implies that the connection lifetime will be the lifetime of the process and that there cannot be more connections than standard channels. Like standard output and standard input, each connection only supports sending data in one direction. Pipes establish this kind of connections. Use the pattern `pipe`.

Sockets

To implement more elaborate patterns of communication, one must be able to create and destroy connections during the execution of a process, and to explicitly choose with whom to communicate. Sockets are used for this, and with sockets, every connection is two-way. Sockets come in two main variants: passive and active. A passive socket is used to define a name, which may be used by active sockets when establishing an actual connection. The interplay is like:

```
Passive: "Here I am! My name is Bob"
...
Active-1: "I want to speak with Bob"
Passive(Bob): "OK, here's a connection"
...
Active-2: "I want to speak with Bob"
Passive(Bob): "OK, here's a connection"
...
Active-3: "I want to speak with Cindy"
(Error: Here's no such thing as "Cindy")
...
```

I.e. active sockets connect by name, and more than one connection may be established by means of one passive socket. The “name” is actually a pair whose first part is an identification of the host (its IP address) and whose second part is an integer (the port number). This pair is unique for each passive socket, at least from the time where the operating system accepts registration of the name until the passive socket is closed. After that, the pair may be reused, that is: the port number may be reused on the given host, if the operating system wishes to do so.

In this library, sockets are also divided along another axis, namely into stream sockets and binary sockets. Stream sockets are specializations of the basic `stream` pattern, and support textual communication. Binary sockets support transfers of blocks of data with a well-known size.

The patterns related to these concepts are: `activeStreamSocket`, `activeBinarySocket`, `passiveStreamSocket`, `passiveBinarySocket` and `socketGenerator`. `SocketGenerators` make it possible to establish more than one connection to one passive socket. `PassiveStreamSocket` and `passiveBinarySocket` are a bit simpler to use, but support only one connection per BETA object.

3.2 The Fragment Communication

3.2.1 The Two Families of Sockets

Basically, `communication` supports two families of sockets: stream sockets and binary sockets.

Stream socket

A stream socket is suitable for transferring data which is readable for human beings, such as the data transferred in a UNIX “talk” session, or the more formal communication between a mail program and an SMTP mail server. A `streamSocket` is a `stream`, so you may “put”, “get” etc. However, do not rely on this kind of socket to transfer data which contains zero-valued bytes, as arbitrary binary data may very well do.

Binary socket

A binary socket is guaranteed to transfer any given block of arbitrary bytes unmodified, but you must always specify the length of the data block, both for sending and receiving.

Both stream sockets and binary sockets come in active and passive versions, and then there are socket generators which are used to generate (stream or binary) sockets whenever somebody tries to connect. These are the main patterns of the fragment, but there are a couple of others as well.

In general, you must have a way of choosing either a binary or a stream variant of a connection to be established, because it is not possible to change a `streamSocket` into a `binarySocket` on the same connection, or vice versa. And each socket object models one connection, so it is not possible to use the same socket object for several different connections - use a fresh object each time instead. For `socketGenerator`, of course, this one-shot-restriction does not apply. See below.

In the following, all the top level patterns in the fragment are described in the order of appearance. After that there is a discussion of how to handle blocking conditions, which applies to all kinds of socket objects.

3.2.2 The Patterns of Communication

`WaitForIO` is used to make this process sleep until some socket "known by `communication`" has data ready for reading, or a formerly full output buffer is no longer full, such that some socket can now be written to. Any OS level socket created by means of `communication` patterns is known, but if you create other sockets, e.g. by using external patterns or by linking with a C-library which creates sockets, they will be unknown. In this case, `waitForIO` may block the process, even though some communication could have taken place.

`AssignGuard` is used to detect wrong usage of other patterns, and `propagateException` is used in error handling. They have no conceptual significance.

Pipe

A `pipe` must be initialized with `init` before usage. Then giving a reference to its `readEnd` (`writeEnd`) as enter parameter to `redirectFromChannel` (`redirectToChannel`) of a not yet started `process` object will attach this `pipe` to another (not yet created) process. If only one end of the `pipe` is attached to another process, the current process may read from (write to) the other end of the `pipe`, when the other process has been created.

StreamSocket and `binarySocket`

`StreamSocket` and `binarySocket` are semi-abstract patterns: It is of no use to create instances of them, but some operations may exit such instances. This is because these patterns implement all of the functionality used during a connection, but they have no means for establishing a connection. To establish a connection, one must choose between playing the passive role or the active role, as described in the preceding section. This concerns the patterns `activeStreamSocket` etc. described below. In the following paragraph the `streamSocket` operations are described in order of declaration.

A `streamSocket` connection may be closed by `close`. After this point, the `streamSocket` cannot be used for communication, so you can discard it. `Flush` ensures that all data in internal buffers of the `streamSocket` actually gets sent. `Put`, `get` and `peek` work as with other streams. `Eos` returns true if no data can be read *right now* from the connection. This is radically different from the semantics of (say) `text`, because with a `streamSocket`, `eos` may be true, and still, at some later point when more data has arrived, become "spontaneously" false. `PutText`, `getLine` and `getAtom` work like in other `streams`.

`NonBlockingScope` is used to handle blocking conditions, and is discussed below.

Error handling in `streamSocket` only discovers that something went wrong, and then terminates the application. To be able to intercept, retry etc. when something goes wrong in a `streamSocket`, use `systemComm` instead of `communication`.

The operations on a `binarySocket` are quite different. These operations are primarily oriented towards transmitting blocks of various kinds of data. In order of declaration:

A `binarySocket` may be `closed`, and is of no use after that. The `writeData` and `readData` operations are used for transferring a block of data given as its starting memory address and the length of the block in bytes. This constitutes the lowest level interface, and as always when using raw addresses: If it is the address of a BETA object, it must be ensured that no garbage collection (GC) can happen from the point at which the address was taken until the point where it is used. This is a bit tricky to ensure because GC happens implicitly. However, only an act of allocation can trigger a GC, so you will be safe as long as no objects are created during the critical period. This means that every object involved in the transfer must be instantiated, and only after that can the address of the BETA object be taken and `writeData` or `readData` executed. `WriteData` and `readData` are already instances in every instance of `binarySocket`.

`endOfData` returns true if no data is immediately available for reading.

getBlock and putBlock

The operations getBlock and putBlock provide support for a very simple, binary data transfer protocol. It supports transfers of blocks of arbitrary length, because the block length is transmitted along with the block itself for the receiver to read. In this protocol, all data is transferred in blocks with the following layout:

```
   len      header        data
|--------|--------|--------------------------------|
```

The len field is a four byte integer value, given in big-endian byte order format. The header field is also a four byte big-endian integer, and it identifies which kind of data is in the data field, what purpose the transfer has, or whatever. The data field length is 4*len bytes long. The sender and the recipient must agree on the interpretation of the header and data fields, which is left unspecified by this level of the protocol.

Operations getBlockLen and getBlockRest are supplied to make it possible for the receiver to read the length of the block to be received, then allocate space for it, and then to receive the block into this space. These two operations will only work meaningfully when used together and in this order.

For all of the operations getBlock, putBlock, and getBlockRest, raw memory addresses are involved, so the same warnings as with writeData and readData apply.

Extended-Repstream

Rising to a more civilized level, the operations putRep and getRep are used to send and receive instances of the pattern ExtendedRepstream. This is a generic container for arbitrary blocks of data, in particular it is possible to put texts and integers into it and read them out again. When receiving data into an ExtendedRepstream with getRep, the ExtendedRepstream will automatically be extended in case the received amount of data exceeds its current capacity.

About nonBlockingScope, see below.

Error handling in binarySocket comes in two levels. At the socket level, you may extend the otherError virtual. This virtual will be executed in response to any error detected during the execution of an operation on the socket, and it is possible to intercept the error by means of a leave imperative in the extending of otherError. Please note that it is not safe to leave from a nonBlockingScope with leave. For this, use leaveNBScope. As otherError is an exception, it will terminate the application unless it is leaved or its continue is assigned the value true.

At the operation level, you may extend error in any operation to take care of errors occurring during the execution of that specific operation. This is the normal way to intercept errors, because it is easy to know which operation went wrong, and this normally influences what is relevant recovery. If error is extended to be leaved, the socket level otherError will not be invoked. One may think of this as a matter of precedence: The operation level error handling has higher priority than the socket level error handling. By default, every communication error terminates the application. By extending, this default may be overridden on each of the two levels.

ActiveStream-Socket

ActiveStreamSocket must have assigned values to its host and port. The host must be given in a format like “quercus.daimi.aau.dk” or “130.225.16.15”. Depending on the network topologi and the whereabouts of this process, some prefixes of the first format may also suffice, notably a format like “quercus”. The port must be an integer. By convention, port numbers below 5000 are reserved for system administration purposes and for special, well-known services like e-mail and ftp. On the other hand, do not expect to be able to use more than a 16-bit unsigned value.

Having initialized host and port, the activeStreamSocket may connect to some existing passive socket, which has been initialized with that port on that host. When connected, it uses the operations of its superpattern streamSocket to communicate, as already described. ActiveBinarySocket is used analogously.

PassiveStream-Socket

For passiveStreamSocket, only port must be assigned. Then bind must be executed to establish (<this host>,port) as a passive socket identifier, to which active sockets may connect. Finally, a connection can be accepted by executing awaitConnection. Again, passiveBinarySocket works analogously. Please note that a passive...Socket can only be used for establishing one connection. If you need to establish more than one connection on some (host,port), use a socketGenerator.

SocketGenerator

The last pattern supplied in communication is socketGenerator. This is a factory from which instances of streamSocket and of binarySocket can be obtained, in response to active sockets connecting to the socketGenerator's port.

As with the passive socket patterns, the port must have some value assigned, and then bind must be executed. To obtain a streamSocket on the next connection requested, execute getStreamConnection, and to obtain a binarySocket, execute getBinaryConnection. As usual, when you are done, execute close on the socketGenerator.

3.2.3 Handling Time with Communication

Often when different processes communicate, it is not possible to predict when data will be available for reading. When writing, a buffer full condition may arise in the kernel of the operating system. Also, accepting a connection from an active socket may happen anytime or never. This means that in most cases, the naive usage of the functionality described in the previous section leads to blocking conditions: The application sits waiting for something to happen, and it cannot do any sensible work in the meantime.

Idle

To remedy this situation, the operations which do not depend on raw memory addresses have an idle virtual, which may be extended to keep the application alive during (possibly) lenghty operations. The idle may be executed one or more times if the operation cannot finish right away. This, however, is not guaranteed to happen so do not rely on idle being executed even once. Do not execute operations on the enclosing socket object within the extending of an idle; this might compromise its internal consistency. Do not stop the operation from within a extending of idle - the operation is unfinished; you may for instance have received half a block, in which case stopping breaks the protocol. Use nonBlockingScope and Blocking for this purpose.

NonBlockingScope

The nonBlockingScope pattern is used for specifying non-blocking communication. This means that operations which cannot begin right away are discontinued. An example is: We try to read from a socket, but no data at all is available to read. If, on the other hand, any irreversible actions have been taken in an operation (e.g. reading a few bytes), it will not be interrupted by the nonBlockingScope mechanism. This means it is always safe to interrupt an operation by enclosing it in a nonBlockingScope, and then later to retry it. It also means that the granularity of scheduling by means of nonBlockingScope is one communication operation; e.g. if the communication partner sends half a block and then takes a break, this process can only execute an idle in the mean time, it cannot switch forth and back between several such ongoing transfers.

With each Idle pattern comes a Blocking virtual. This is executed if the current operation is blocking, i.e. if nothing can be done right away and nothing has been done yet. You may extend this virtual to take some action in response to the operation being blocked. If the operation is enclosed in a nonBlockingScope, Blocking gets executed immediately before the operation is interrupted. If you do not want to interrupt the operation, execute continue in a extending of Blocking.

As default, the communication will be blocking. But if you enclose an operation in a specialization of nonBlockingScope, we leave the nonBlockingScope at the first blocking condition. *Please notice* that it is unsafe to execute a leave statement which leaves a nonBlockingScope. If you need to explicitly leave it, execute leaveNBScope. The normal usage without and with nonBlockingScope looks like this:

```
(* BLOCKING STYLE *)
myStreamSocket.getLine (* waits until data has arrived *)
 -> reactOnInput;    (* always executed *)
reactSomeMore;     (* always executed *)
doOtherThings;

(* NONBLOCKING STYLE *)
myStreamSocket.nonBlockingScope
  (#
  do
   myStreamSocket.getLine (* if no data: leave scope at once *)
    -> reactOnInput;    (* only executed if data available *)
   reactSomeMore;     (* only executed if data available *)
  #);
doOtherThings;
```

With some operations such as `writeData` and the like it is not possible to have a virtual `Blocking` or `Idle` pattern, because they depend on raw memory addresses. However, enclosing such operations in a `nonBlockingScope` does indeed cause them to behave in a non-blocking manner. Having stopped such an operation because it threatened to block, the raw memory address will have to be recomputed before the operation is retried (assuming it is the address of a BETA object).

3.3 The Fragment systemComm

The fragment `systemComm` provides a functionality similar to that of the fragment `communication`, but it is in several ways more sophisticated. Any program using `systemComm` must be a `systemenv` program, because `systemComm` heavily depends on cooperation with the scheduler present in `systemEnv` programs.

BETA co-routines

Instances of the patterns of this fragment are expected to be executed from BETA co-routines, and such co-routines must tolerate being suspended (de-scheduled) and later re-scheduled as part of the execution of possibly lengthy `systemComm` operations. This means that concurrency control by means of `semaphores`, `monitors`, and the like must be established almost as rigourously as had the co-routines been fully concurrent threads of execution.

In return for this increase in complexity, a usually very important reduction in complexity arises from having implicit instead of explicit scheduling. Especially when fitting a new piece into an existing framework it is a great asset to be able to simply “spawn” the new piece as part of an initalization phase and then have it running along with the rest of the program without changing any of the other parts not directly interacting with this new piece.

In more concrete terms, it works like this: Whenever an operation is about to block, the current component will be suspended. It will be resumed some time later, when the requested IO is available. In the meantime, some other component which has requested IO available or is not waiting for IO will be resumed. In this way the following liveness property of the program is ensured: it will never be the case that a `systemComm` operation by blocking delays the continuation of the execution of all of those components which are either (1) not executing a `systemComm` operation or (2) executing a `systemComm` operation, but has IO of the requested kind available. Of course, any component can still block the whole system by, for example, entering an infinite loop that does nothing.

`SystemComm`, like `communication`, supports the two families of sockets: stream sockets and binary sockets. Everything said in section 3.2.1 still holds in the context of `systemComm`.

The following section describes the top level patterns of systemComm in order of appearance. After that, there is a section with a general discussion of error handling, which applies to all parts of systemComm. Finally another section discusses the treatment of timeout. This again applies to all of systemComm.

3.3.1 The Patterns of SystemComm

WaitForever is a constant used to specify an infinite timeout.

AssignGuard is used to detect wrong usage of other patterns, and propagateException is used in error handling. None of them are important for the understanding of the fragment.

Pipe

A pipe must be initialized with init before usage. Then giving a reference to its readEnd (writeEnd) as enter parameter to redirectFromChannel (redirectToChannel) of a not yet started process object will attach this pipe to another (not yet created) process. If only one end of the pipe is attached to another process, the current process may read from (write to) the other end of the pipe, when the other process has been created.

StreamSocket and binarySocket

StreamSocket and binarySocket are semi-abstract patterns: It is of no use to create instances of them, but some operations may exit such instances. This is because these patterns implement all of the functionality used during a connection, but they have no means for establishing a connection. To establish a connection, one must choose between playing the passive role or the active role, as described in section 3.1. This concerns the patterns activeStreamSocket etc. described below. The following describes the operations of streamSocket in order of appearance.

SameConnection

The operation sameConnection on a streamSocket is used to check whether two different instances of streamSocket are attached to the same operating system level socket. This may happen if one streamSocket is created and a connection is established, and then later this connection silently gets destroyed. Now it is possible to establish a new connection with a new streamSocket instance, and to get from the operating system the same connection identifier (file descriptor) as was used by the first connection. In this case, the first streamSocket will happily communicate on the NEW connection, giving rise to strange errors: It suddenly talks with some total stranger, as far as the original purpose of this streamSocket is concerned.

getPortable-Address

The getPortableAddress operation is used to obtain an instance of a portableCommunicationAddress which describes this passive socket or describes the destination of this active socket, whichever variant is at hand. A streamSocket connection may be closed by close. After this point, the streamSocket cannot be used for communication, so you can discard (i.e. forget) it. Flush ensures that all data in internal buffers of the streamSocket actually gets sent. Put, get and peek work as with other streams.

Eos returns true if no data can possibly be read from this connection now or ever. Please note the difference from the semantics of the communication version: This semantics more closely resembles the semantics of eos on other streams. On the other hand, it may still happen that the communication partner holds the connection alive but will not write any more data to it. In this case, this process has no chance of guessing that no more data will actually arrive, so eos will “spontaneously” change from false to true when the other process actually closes the connection. This “spontaneous” change goes in the opposite direction as the one in the communication version.

PutText, getLine and getAtom work like in other streams. ForceTimeout is used to provoke the same response within an ongoing operation as would have been the result of a timeout. This makes it possible to exercise timeout control over an operation from within a co-routine different from the one executing that operation. Moreover, it makes it possible to define a timeout limit for the execution of a number of operations, instead of setting timeouts for each of them. UsageTimeStamp returns an integer value which indicates when this socket was last used. The value makes sense only when compared to usage time stamps of other sockets in this same process. The purpose is to enable a user of many sockets to close the least recently used connection or similarly when and if the process runs out of system resources (e.g. it experiences a “to many open files” error).

`NonBlockingScope` and `leaveNBScope` are used to handle blocking conditions. The discussion given on this in section 3.2.3 applies without change. But this approach has been made largely obsolete by the implicit scheduling built into every operation in `systemComm`. An exception is the possible usage of an `idle` virtual to keep some kind of progress feed-back running, thus reassuring the user that the communicating thread of execution has not gone into oblivion.

The operations on a `binarySocket` are oriented towards transmitting special generic containers for blocks of arbitrary bytes. Comparing with `communication`, operations depending on raw memory addresses at the interface level are no longer present.

`SameConnection` and `getPortableAddress` work analogously to the operations with the same names in `streamSocket`.

As always, `close` a socket when done with it. `endOfData` is true if no data is immediately available for reading. Please note that this semantics may be updated to resemble the semantics for `eos` with `streamSocket` in a later release. In the context of implicit scheduling, the current semantics is of little use.

PutRep and GetRep

`PutRep` and `getRep` are used to send and receive instances of the pattern `ExtendedRepstream`, and `putRepObj` and `getRepObj` are used to send and receive instances of the pattern `RepetitionObject`. The protocol for transmitting `RepetitionObjects` is a little different from the one used with `ExtendedRepstream` objects: there is no header field, and the length field is the first element in the repetition from the `repetitionObject`, i.e. `repetitionObjects` have their length “built-in”.

```
  len      data
|--------|--------------------------------|
```

Otherwise, it is like the protocol for `ExtendedRepstream` objects.

`ForceTimeout` and `usageTimestamp` work as described with the corresponding `streamSocket` operations.

Again, the discussion about handling time and blocking conditions given in section 3.2.3 applies to `nonBlockingScope` and `leaveNBScope` here. And again: it is largely obsolete, as pointed out above in relation to the same patterns in `streamSocket`.

ActiveStream-Socket

`ActiveStreamSocket` must have assigned values to its `port` and to at least one of its `host` and `inetAddr` attributes. In case both `port` and `inetAddr` are assigned a value, `inetAddr` takes precedence.

The host must be given in a format like “quercus.daimi.aau.dk” or “130.225.16.15”. Depending on the network topologi and the whereabouts of this process, some prefixes of the first format may also suffice, notably a format like “quercus”. The port must be an integer. By convention, port numbers below 5000 are reserved for system administration purposes and for special, well-known services like e-mail and ftp. On the other hand, do not expect to be able to use more than a 16-bit unsigned value. The value to use when assigning `inetAddr` must be the four-byte internet address, given as an integer value. E.g. the absolute address “130.225.16.15” is given as the integer 2195787791.

This done the `activeStreamSocket` may `connect` to some existing passive socket, which has been initialized with that port on that host (with that internet address). Having connected, it uses the operations of its superpattern `streamSocket` to communicate, as already described. `ActiveBinarySocket` is used analogously.

PassiveStream-Socket

For `passiveStreamSocket`, only `port` must be assigned. Then `bind` must be executed to establish the given port number as an address, to which active sockets may connect. Finally, a connection can be accepted by executing `awaitConnection`. Remember to enter a timeout value to `awaitConnection`. Again, `passiveBina-`

`rySocket` works analogously. Please note that a `passive...Socket` can only be used for establishing one connection. If you need to establish more than one connection on a given port, use a `socketGenerator`.

A `socketGenerator` is a factory from which instances of `streamSocket` and of `binarySocket` can be obtained, in response to active sockets connecting to the `socketGenerator`'s `port`.

As with the passive socket patterns, the `port` must have some value assigned, and then `bind` must be executed. To obtain a `streamSocket` on the next connection requested, execute `getStreamConnection`, and to obtain a `binarySocket`, execute `getBinaryConnection`. As usual, when you are done, execute `close` on the `socketGenerator`.

`getPortableAddress` exits a `portableCommunicationAddress` which describes the network identity of this `socketGenerator`. `ForceTimeout` and `usageTimeStamp` work as with the other socket variants, and the considerations concerning `nonBlockingScope` and `leaveNBScope` are as usual.

3.3.2 Error Handling in SystemComm

Throughout `systemComm`, the facilities from the fragment `errorCallback` are used in the handling of errors.

3.3.2.1 Error Callbacks

An error callback is a virtual pattern which is invoked in response to the occurrence of some error. Whenever an error condition is detected on a socket, a corresponding

virtual pattern is instantiated and executed. These patterns are specializations of `errCB`, as declared in `errorCallback`. Such virtual patterns are hereafter denoted error callback patterns. To catch and treat an error, extend the corresponding error callback.

If an error callback is not extended and the corresponding error occurs, an exception is executed and the program terminates. If the error callback is extended, the following holds:

- if `abort` is executed in the extending dopart, the operation (but not the program) is aborted. You may execute `leave` within a specialization of abort. Do not `leave` an error callback from any other point, as this may put the object or the process into an unstable state. If you `abort` but do not `leave`, the operation aborts, but control flow is like when the operation succeeds; in this case, any exited values are dummy values, reflecting that the operation failed. Do not use them! Actually, do not `abort` without `leave`!
- if `continue` is executed in the extending dopart, there will be an attempt to recover and finish the operation after the execution of the error callback terminates. For many types of errors, no general recovery is possible at the operation level. But you could close a couple of files in response to a `resourceError` and then execute `continue`. In case of timeout, you can always choose to take another turn with `continue`.
- if `fatal` is executed in the extending dopart, an exception will be executed and the program will be terminated. So the execution of the error callback will not return. This is also the default, but with hierarchical error callbacks, you may need `fatal` to undo a `continue` at a higher level.

In case it happens more than once that an operation from the set {`abort`,`continue`,`fatal`} is executed, the one executed as the last takes precedence.

3.3.2.2 Error Propagation

As mentioned, the error callback patterns are present at three different levels: Concrete error callbacks, operation level error callbacks, and socket level error callbacks.

The concrete error callbacks provide the greatest level of detail: their names indicate the kind of error condition detected. This makes it possible to treat different errors differently.

The operation level error callback is executed whenever an error condition is detected during the execution of that operation. In a extending of this kind of error callback, you can adjust the default action for all the concrete error callbacks in this operation. The single socket level error callback is executed whenever any operation detects any error condition. In a extending of this error callback, you can adjust the default action for all concrete and operation level error callbacks.

The means for adjusting the behaviour is in all cases to execute `abort` (probably `abort(# leave L #)`), `continue`, or `fatal`, and the semantics of these imperatives are the semantics of the concrete error callbacks described in section 3.3.2.1.

Error callback extendings take precedence like this, in ascending order: concrete level, operation level, socket level. This means that the higher level specifies a default, and the more concrete level may override this default by executing `continue`, `abort`, or `fatal`.

3.3.2.3 Categories of Errors

At the concrete level of error callbacks, errors are categorized according to classes of operating system level error messages.

The list of names used for concrete error callbacks and a short description of the corresponding class of operating system level error is as follows:

Error callback name	Meaning
accessError	insufficient access rights
addressError	address (i.e. (host,port)) in use or invalid
badMsgError	(EBADMSG, hardly documented in man page)
connBrokenError	connection has become unusable
eosError	unexpected end-of-stream
getHostError	error when getting hostname
internalError	should not happen; please report if it does!
intrError	operation interrupted by signal
refusedError	connection refused by peer
resourceError	too few file descriptors/buffers etc.
timedOut	specified timeout period has expired
timedOutInTransfer	timed out, and some data have been
transferred	
unknownError	OS reports unknown errno (new OS?)
usageError	e.g. you must initialize port before connecting
accessError	(streamSocket) as above
nospaceError	(streamSocket) caused by lack of resources
readError	(streamSocket) error during read operation
writeError	(streamSocket) error during write operation
otherError	(streamSocket) anything else

(In the case of `streamSockets`, the errors are currently not being categorized so precisely as they should. These errors are given in the last five entries of the table and are marked with “(streamsocket)”. They will very likely be refined into the first 14 categories of the table in a future release of this software).

3.4 Timeout Management

Because most operations in `systemComm` may provoke the suspension (de-scheduling) of the current co-routine, any such operation may implicitly prevent this co-routine from making any progress for an indefinite period of time. To give the co-routine the power to do something about this, each of these operations takes a specification of an upper limit (in seconds) to the time elapsed during the execution of that operation.

When such a timeout has been specified for some operation, the scheduler will resume the execution of that operation if it gets the control and the timeout period has expired. This means that lots of activity in the system as a whole may postpone the detection of a timeout somewhat, and - as usual - an infinite loop somewhere could stop everything.

In practical terms, the operation is resumed when and if the timeout period expires, and of course it resumes by executing an error callback. Two different error callbacks may be used to indicate the problem. If no irreversible actions have been taken, the `timedOut` error callback is used. If some irreversible actions have been taken, such as receiving or sending part of a message, the `timedOutInTransfer` error callback is used. This last situation is considerably more grave than the first: Aborting an operation "in-transfer" means breaking the protocol, which again means that any subsequent messages received on the same connection will be garbled. Resynchronization is hardly possible unless the data transferred are lines of text or some other format with built-in structural markers. So in this situation, give it another chance, or close the connection.

For `streamSocket` and its subpatterns, the socket level attribute `timeoutValue` decides the timeout for all operations. For `binarySocket` and its subpatterns, each operation which has timeout control takes the timeout value as its first enter parameter. Likewise with `socketGenerator`. If you forget to specify such a timeout value, e.g. in `awaitConnection` on a passive socket, the operation will always terminate at once with a timeout error.

4 Addresses

The fragment `commAddress` supports representing addresses of communication ports with which one might like to establish connections. In this setting, more different operating systems and kinds of communication ports are covered than what `communication` and `systemComm` actually support as yet. Accordingly, TCP/IP sockets are just one example of a kind of communication port.

Instances of any of these patterns are values, and under normal circumstances their identity will make no difference. This ensures that it makes sense to translate them from BETA objects into simple strings of text and back again, and this eases the migration of such values across networks and other media.

At the most abstract level, `portableCommAddress` models a portable communication address. This specifies the address of a single destination or the address(es) of a group of destinations.

The patterns `portableMultiAddress` and `portablePortAddress` specialize `portableCommAddress` into concrete patterns for the multiple-destination case and one-destination case, respectively.

The pattern `concretePortAddress` and its specializations represent non-portable, protocol specific communication port addresses. Of course, any `concretePortAddress` is portable, being a normal BETA object; but only on some platforms will it be possible to have such a communication port as is specified by the `concretePortAddress`.

`ConcretePortAddresses` are kept in `portableCommAddresses` and selected according to protocol specifications, given as `protocolSpec` objects.

4.1 Specification of Connection Requirements

The pattern `protocolSpec` is used to package a specification of requirements to a communication transfer. This package is given to a `portablePortAddress`, which will then use it to choose an appropriate channel. A specification is built with an instance of `protocolSpec` by setting its `cType` and `rType` attributes. For these, choose from the constant values given in the fragment `commError`.

The `cType` value can be any of the constants `commProtocol_...` and specifies that the chosen channel must be a TCP/UDP/etc. connection or that any kind of connection will do (`commProtocol_dontcare`).

The value of `rType` is any of the constants `commRely_dontcare` (no requirements), `commRely_unreliable` (allow all the below mentioned kinds of malfunction) or `commRely_reliable` (prevent all those malfunctions). Or it is a sum of some of the constants `commRely_loss` (prevent packet lossage), `commRely_dup` (prevent packet duplication), `commRely_order` (prevent packets from arriving out of order), `commRely_contents` (prevent packets from having corrupt data).

In reality, the last guarantee is enforced by means of checksums or something similar, so it is only very unlikely that a packet with corrupt data will pass unnoticed, not impossible. Moreover, all the other guarantees depend on having packets with trustworthy (header) contents, so not all combinations make sense.

4.2 The Abstract Level

The abstract pattern `portableCommAddress` is used to specify the identity of an abstract communication address. The patterns `portableMultiAddress` and `portablePortAddress` are its non-abstract specializations.

Before usage, initialize any specialization of `portableCommAddress` with `init`.

Any `portableCommAddress` is able to express its value in textual form, by the operation `asText`. This enables simple and safe migration of an instance of any specialization of `portableCommAddress`: Translate it into text, send it across the network, write it into a disk file, or whatever, and then reconstruct it as a BETA object from its text value.

Tell a `portableCommAddress` what proporties are required of the communications associated with it by entering a `protocolSpec` object reference. This affects its choice of concrete communication port(s) in subsequent communications.

To reconstruct a `portableCommAddress` from its text representation, give it as enter parameter to `portableCommAddressFromText`, and a corresponding object will be exited. The text is expected to have been produced by some instance of a specialization of `portableCommAddress` using its `asText`.

Problems in this process are reported by invoking `parseError`. This terminates the application, unless you extend `parseError` to handle it.

4.3 The Concrete Level

A `portableMultiAddress` specifies a group of communication ports. Start or enhance the group by `insert`ing members. Reduce it by `delete`ing members.

A `portablePortAddress` specifies the identity of one logical communication destination. A logical destination corresponds to a number of concrete communication ports, represented by instances of specializations of `concretePortAddress`. It is up to the user of these patterns to ensure that the contained set of concrete ports actually "logically belong to the same destination".

The idea is that if "I" can talk on a channel of type "{A,B}" and "you" can talk on a channel of type "{B,C,D}", it is up to the underlying framework to discover that in order to establish a connection, "we" must use type "B".

A `portablePortAddress` can be built by inserting specializations of `concretePortAddress`. Only one concrete address is allowed for each known type - inserting a second instance overrides the previously inserted one. With `delete`, any concrete port can be removed again. To retrieve a concrete port (without removing it), use one of the `Get...Port` operations. If this `portablePortAddress` does not contain any concrete port of the requested variety, `NONE` is exited.

`ConcretePortAddress` is an abstract superpattern for specifying the address of a concrete communication port, such as a UNIX stream socket, a Macintosh PPC ToolBox session, a shared memory buffer etc.

Like a `portableCommAddress`, each concrete specialization is able to express its value textually with the operation `asText`, and it is able to characterize its communication protocol with the operation `protocol`. The operation `protName` exits a text which is a short, descriptive name for that protocol, and `conformsTo` answers true/false to the question, whether this kind of connection conforms to the protocol associated with an entered `commProtocol_...` constant.

The pattern `unixAbstractPortAddress` captures similarities between TCP and UDP ports, represented by `tcpPortAddress` and `udpPortAddress`. The `tcpPortAddress` also fits a MacTCP port. The pattern `unixPortAddress` represents an AF_UNIX address family socket, i.e. it appears as a name in some directory, just like a file; `ppcPortAddress` represents a Macintosh PPC ToolBox session; `memPortAddress` corresponds to a shared memory implementation of inter-process communication.

5 Managing a Pool of Connections

A connection pool manages a number of client side communication interfaces (e.g. active sockets), and allows choosing which one of them to use for a communication transfer by means of a `portableCommAddress`. This abstracts away the need to establish connections: whenever a connection as specified is available in the pool, we use it. Otherwise, such a connection will implicitly be established and added to the pool. If this process runs out of resources associated with these connections (e.g. file handles), it is possible to ask the pool to close the least recently used connection.

Concurrency control

The connections are subject to concurrency control, so they must be used in a "take-it, use-it, give-it-back" fashion. This is achieved by the pattern `communication`. The concurrency control is necessary to prevent the situation where two users of the pool both transmit messages to some other party on one given connection, and randomly divide the incoming messages on that connection between them, both believing to have the other party for themselves. Using the pattern `communication`, at most one user of the pool communicates on any given connection at any given point of time.

Binary socket connections

By now, the only variant of connection pool implemented is the `binaryConnectionPool`. Instances of `binaryConnectionPool` are used for managing a number of binary socket connections. Before usage, `initialize` it. The user of a `binaryConnectionPool` gives a specification of the receiver, the type of connection, the quality of service etc. in a `portableCommAddress` to a (specialization of) the control pattern `communication`. This is used as follows (where `bcPool` is an instance of `binaryConnectionPool`):

```
addr[] -> bcPool.communication
(# (* Extend error callbacks here *)
do
  (* Within this dopart: use 'sock' to communicate *)
  (* Do not bring references to sock outside *)
#);
```

If you want to `leave` the dopart of a specialization of a `communication`, use a construction like `leaving(# do leave L #)` in stead of `leave L`. Otherwise some resources may be rendered inaccessible.

Whenever the pool establishes a new connection, the hook `onNewConnection` of `communication` is executed. In a extending of this hook, a reference to the newly established connection is available, and by assigning a co-routine to `actor`, the connection gets associated with this co-routine. This is used to handle incoming messages to connections in the pool, which are not the immediate response to an outgoing message transmitted in a usage of `communication`: have the co-routine sit around waiting for the incoming messages. To support such things, one must specialize `binaryConnectionPool`.

If the connection delivered as `sock` within a specialization of `communication` is to be taken away from the pool and used outside, execute `removeSock` and bring out a reference to `sock`. If it is known that the connection will not be useful anymore, execute `removeSock` and `sock.close`.

On exceptions, see the description in section 3.3.2.

The operation `markAsDead` is used to tell the pool that it certainly cannot have a connection like the one entered. If a communication partner closes a connection (or perhaps terminates unexpectedly), and the other end of that connection is in a connection pool, it could happen that this connection is not chosen in any `communication` for some time. If a new connection is created, the operating system may then reuse the local connection identifier (file handle, in case of UNIX sockets), giving a totally different connection, which is then administrated by some new BETA socket object. Now two BETA socket objects will talk to the same OS level connection (file handle), but this means that the first object (in the pool) has silently been “redirected” to a new communication partner. Of course, this leads to strange errors.

So, whenever creating a BETA socket object OUTSIDE a connection pool, please tell it by means of `markAsDead`, that any connections in the pool with the same OS level identifier must have died silently and thus should be removed from the pool. Internally, the connection pool handles this automatically.

Please note that this problem is not specific for connection pools, for the `process` library, or even for BETA programs, for that matter. But it occurs mainly in the presence of complicated and very dynamic communication topologies, which are more likely to appear with connection pools. It would actually be best to carry out similar checks (using `sameConnection`) also when using only simple socket objects in an application.

`removeSomeConnection` will seek through all unused connections in the pool. An unused connection is a connection such that no instance of `communication` in any coroutine of this process currently refers to it with its `sock` attribute. From this set of unused connections, it chooses the least recently used (as reported by its `usageTimestamp`), closes it, and removes it from the pool. If all connections are currently in use, application specific actions must be taken to free some of them. The callback `noConnectionsRemovable` is executed in this situation. It does not terminate the application by default, so beware of the possible infinite retry loop if `removeSomeConnection` is used in response to `resourceError`, and no connections could actually be removed.

When done with a `connectionPool`, `close` it to close all of the connections contained within it.

6 Managing co-routines

The fragment idScheduler uses neither processmanager, communication nor systemComm, so in a way it is an island of its own. It typically comes together with the other parts of the process library when a communication connection is shared by a number of BETA co-routines. In this case, a (master) co-routine administrating the connection must have some means to control the execution of the (slave) co-routines using the connection. This means the slaves must be able to "suspend" themselves wrt the master, and the master must be able to "resume" a slave when the connection has data ready for it.

As usual when present, the init operation should be executed on each instance of idScheduler before first usage.

Instances of idScheduler can play this role as an "intermediate" scheduler, controlling any number of co-routines. Each slave co-routine may id_suspend itself, awaiting an event identified by the integer value id entered. The id_suspended slaves are under the control of the idScheduler master, and the master may resume slaves by executing id_resume, again choosing which slave to wake up in accordance with the id entered.

These id values must be unique for the whole set of possible users of any given idScheduler. Otherwise the semantics will be quite different from what is described here. Usually, one can use a global "id-factory", which always delivers new, essentially meaningless values. In particular, it is a bad idea to use values which are constrained by other parts of the application ("have a meaning"), because such constraints may one day force some ids to have the same value.

id_suspend and Id_resume

In the following, an instance of id_suspend and an instance of id_resume are called corresponding if their id items have the same value; an id and a slave are corresponding if the slave is id_suspended and the id_suspend.id equals id; similarly for other combinations.

Add further attributes to the isElement virtual to create holders of information transferred from the master to the slave when the slave is resumed. A specialization of id_resume may for instance transfer information to a corresponding slave by assigning some object reference to a dynamic reference item, say "info", in its elm. When the corresponding slave wakes up, its id_suspend.elm.info will refer to that object. (For a concrete example, check out demo/idSchedulerDemo.bet, where this technique is used to transfer a text).

The specialization idTimeoutScheduler allows a slave to specify a timeout limit to the period of suspension, using the operation id_timeoutSuspend. This operation matches id_resume (there is no need for an id_timeoutResume).

If a period of length timeoutvalue expires after a slave has id_timeoutSuspended itself with no occurrence of a corresponding id_resume, the slave virtual id_timeoutSuspend.retry gets to decide whether or not the suspension should be continued. If yes, another period of waiting starts. If no, the onTimeout callback is executed, and that ends the id_timeoutSuspend. (Actually onTimeout does NOT get executed -- please refer to section 8).

If, on the other hand, the corresponding id_resume does occur within the timeout period, the slave callback id_timeoutSuspend.onSuccess is executed, and that of course also ends the id_timeoutSuspend.

Now, if the master administrates a (number of) connection(s), the slaves can share it (them) in the following way: The `id` values used can be described as transactions identifiers, and these transaction identifiers are transferred along with other data across the network. Now, a slave can acquire access to a connection to send a request "`D`", and then `id_suspend` itself on the transaction identifier. Each time data can be received on a connection, the master reads the transaction identifier and then `id_resume`s the corresponding slave, probably providing this slave with access to the connection by means of the "`elm.info`" technique described above. Now, the slave can use the connection to collect the answer to the original request "`D`". In the meantime, many other slaves could have sent and/or received data on the same connection -- and, importantly, the slaves do not have to know about each other. As the (set of) connection(s) is a shared resource, there will have to be some concurrency control associated with it.

7 The Demo Files

A number of demonstration files are provided in the subdirectory `demo`. They show simple and typical ways to use the process library. The files generally use `communication`, so some transformations will be needed in order to use them with `systemComm`.

Because of the "process" aspect, and because of the nature of inter-process communication, the demo files come in small groups. For some groups, one program will manipulate others. For other groups, one may start a "server" and some "clients" and then interact with the clients to initiate communication. In the following, the groups are presented one by one.

7.1 activate

This is a stand-alone demo which uses a `process` to start the BETA compiler and a `pipe` to tell it to compile some fragment named `betaProgram`. You may have to create such a fragment. Please note: the released version of this demo is incorrect. Refer to section 8 which lists a better one.

7.2 pipeline, consumer and producer

Execute `pipeline, which will then start producer and consumer` in such a way that standard output from `producer` is piped into standard input of `consumer. The file items is read in by consumer` and written to its standard output.

7.3 exchange

Starts an executable `igor` which is given the argument `rottweiler` by means of `process.argument.append.` Then, while igor is running, exchange prints out a small message every few seconds. When stops, `exchange` also stops (after the termination of the current delay period). One could for example do:

```
cd <<my directory for trying out little things>>
cp /users/beta/process/v1.4/demo/exchange.bet .
cp exchange.bet rottweiler.bet
ln -s /usr/local/lib/beta/bin/beta igor
beta exchange
./exchange
```

The `exchange` executable is of course a CPU hog, because it sits in a tight `for` loop during those few seconds of delay.

7.4 firstProgram and otherProgram

When executed, `firstProgram will start otherProgram and accept a streamSocket` connection from `otherProgram`. Then they exchange a couple of words, and both terminate.

7.5 aClient and theServer

When `theServer` is executed, it starts two instances of `aClient` and communicates a little with them over two `streamSockets`, one for each client.

7.6 aBinClient and theBinServer

Very similar to `theServer, theBinServer starts two instances of aBinClient` and communicates with them. This time, `binarySockets` are used, and blocks of arbitrary bytes are being transferred. Of course, the data transferred is just a usual BETA integer, but there is no essential difference to the case where any other block of memory is transferred.

7.7 aRepClient and theRepServer

Using a similar setup, but extending the preceding two demo groups a bit, `theRepServer and theRepClient` communicate according to a small, higher-level protocol. Generic containers for blocks of bytes, namely `extendedRepStreams`, are used for the transfers. The protocol specifies three different formats for the contents of these `extendedRepStreams`, distinguished by the tag value `header`, which is transferred along with the `extendedRepStream in the binarySocket operations putRep` and `getRep`. It should be fairly easy to read the exact protocol out of the fragment `showRep`.

7.8 chatClient and chatServer

This group is used interactively. Start `chatServer` and then a number of instances of `chatClient`. Each client will connect to the server, resulting in a star-shaped connection topology. One may interact with each of the clients, and the clients in turn interact with the server.

The fragment `commandCategory` is used to distinguish different types of commands. The command language is very simple: anything starting with the letter “q” is a Quit command, anything starting with an “a” is an Answer command, and anything starting with an “A” is an AnswerWait command. Anything else is a Default command. Enter commands as any piece of text at the prompt, ending with `RETURN`. Please note that leading whitespace is significant.

All commands are immediately forwarded to the server. Then, if the command was a Quit command, the client closes down the connection and terminates. If it was an Answer command, the client notifies the user of that fact by printing a message containing the sequence number of this Answer command. Some time later, the server will return an answer, and the sequence number of the answer makes it possible to match up outgoing requests with incoming answers. In case of an AnswerWait command, the client blocks until the answer from the server arrives. For Default commands, the contents are just echoed at the server.

For each command received, the server echoes the identification number of the client which sent that command and the contents of the command. You may wish to examine the source code in `chatServer.bet to see how nonblockingScope` enables the server to (semi-)simultaneously receive incoming messages, accept connections from new clients, and do other work.

7.9 repChatClient and repChatServer

Similar to `chatClient` and `chatServer`, using `binarySockets` for the communication.

7.10 idSchedulerDemo

This demo shows a simple application of an `idTimeoutScheduler` which uses neither `processmanager`, `communication`, nor `systemComm`.

`An instance of idsched_master` plays the “master” role, and a number of `idsched_slaves` play the “slave” role, as described in section 6. Each slave has an identifier, which is also used as the timeout period in its `id_timeoutSuspend operations.`

`When idSchedulerDemo` runs, a master and a number of slaves are created. The number of slaves is specified as the first command line argument. The master immediately goes to sleep, and sleeps for as many seconds as the second command line argument specifies. The slaves start `id_timeoutSuspending` themselves, allowing two retries. By the third retry, a slave will give up and terminate (the comment “Give up at second attempt” in `idSchedulerDemo.bet` is misleading). When the master wakes up, it serves the slaves in order.

Try `idSchedulerDemo 2 3` to watch a small but non-trivial case; try `idSchedulerDemo 20 30` to get a feeling for the behaviour at a somewhat larger scale.

8 Known Bugs and Inconveniences

In systemComm, the streamSocket operation eos does not correctly implement the described semantics. Errors in system calls are detected as they should be, and the answer is correctly "false" when data is immediately available, but when data is not immediately available, the return values are swapped: When the communication partner has closed down the connection, the answer will be "false", and when this has not happened, the answer will be "true". A patch to fix it is to swap the lines 494 (// 1 then) and 498 (// 0 then) of private/ssocket_unixbody.bet. There is no known easy workaround.

For streamSockets in both communication and systemComm, reading a line of text with the operation getLine or a word with getAtom only works correctly when the line/word becomes available to read as a whole. If a non-empty part of the line/word but not all of it can be read, the operation incorrectly detects an error. A possible workaround is to use get and collect characters in a normal BETA text object, on which getLine and getAtom can be used.

If the transmitting side always sends lines/words in one go, the problem is unlikely to show up. In this case, if the purpose is non-critical of course, you could try to ignore the problem.

Outputting operations in streamSocket, such as put, flush and putLine, will not detect a buffer full condition before attempting to transmit data. This means that they may block until the operating system has relieved the full buffer of some of its contents. This usually happens quickly, though.

Certain operations in systemComm take as enter parameter a timeout value, which does not affect the execution of the operation, because timing out makes no sense - the operation is not "possibly lenghty". An example is close of binarySocket.

Furthermore, the timeout enter parameter in the streamSocket pattern withPE provides the operations open, close and flush with such an enter parameter, and this is no longer used. As described, timeoutValue is used to specify timeouts in all streamSocket operations.

In portableMultiAddress, members are deleted by identity, i.e. entering a reference to some portablePortAddress in an invocation of the delete operation will delete that exact instance, if present. It would make more sense to delete every portablePortAddress contained by this portableMultiAddress, which specifies the same communication port as the one entered. That is, it would be better if members were deleted by value equality.

portableMultiAddress ought to have means for iterating through all its members, such as a scan operation. There should also be a way to test for equality and for subset-relations between portablePortAddresses, and between portableMultiAddresses.

In the fragment connectionPool, in the pattern communication in binaryConnectionPool, the operation removeSock does not remove the connection denoted by sock as it should. Workaround: Use sock[]->markAsDead whereever removeSock should have been used.

The demo-file activate.bet is garbled. Use the following instead:

```
ORIGIN '~beta/process/v1.4/processmanager';
--- program:descriptor ---
(#
   compiler: @process;
   aPipe: @pipe;
do
   '/usr/local/lib/beta/bin/beta'->compiler.init;
   aPipe.init;
   aPipe.readEnd[]->compiler.redirectFromChannel;
   compiler.start;
   'betaProgram'->aPipe.writeend.putText;
   aPipe.writeend.newLine;
#)
```

where `betaProgram.bet` is the a path of some BETA source code file.

In the fragment `idScheduler, the callback onTimeout of the operation id_timeoutSuspend` is never executed, even though it should be executed in case of a timeout. For a workaround, put the code intended to go into a specialization of `on-Timeout` at the end of a specialization of the `retry` virtual, and encapsulate it within an `if` statement such that it is executed if `retry` exits false.

9 Interface Description

9.1 commAddress

```
(* CONTENTS
 * ========
 *
 * Defines patterns for representing communication addresses.
 *
 * The most abstract pattern, portableCommAddress, models a
 * portable communication address. This specifies the address
 * of a single destination or the address(es) of a group of
 * destinations.
 *
 * The patterns portableMultiAddress and portablePortAddress
 * specialize portableCommAddress into concrete patterns for
 * the multiple-destination case and one-destination case,
 * respectively.
 *
 * The pattern concretePortAddress and its specializations
 * represent non-portable, protocol specific communication
 * port addresses. These are kept in portableCommAddresses
 * and selected according to protocol specifications, given
 * as protocolSpec objects.
 *)

(* Specification of connection requirements
 * ========================================
 *
 * Used to package spec. of requirements to a communication
 * transfer, and then given to a portablePortAddress, which
 * will use it when choosing an appropriate channel.
 *)
protocolSpec:
  (#
     cType: @integer; (* one of 'commProtocol_.*'; dontcare is default
*)
     rType: @integer; (* one of 'commRely_.*'; dontcare is default *)
     (* bandwidth/r-rr-rra/etc *)
  enter (cType, rType)
  exit cType
  #);

(* Portable communication address
 * ==============================
 *
 * Specifies identity of an abstract communication address.
 * This pattern is abstract, and no instances of it are
 * expected to exist. The patterns portableMultiAddress and
 * portablePortAddress are non-abstract specializations.
 *
 * Any portableCommAddress is able to express its value
 * in textual form, by 'asText'.
```

```
 *
 * Tell a portableCommAddress what proporties are required
 * of the communications associated with it by entering
 * a protocolSpec object. This affects its choice of
 * concrete communication port(s) in subsequent
 * communications.
 *)
portableCommAddress:
  (#
     init:< Object;
     asText: @asTextPattern;

     (* private *)
     asTextPattern:< (# t: ^text do INNER exit t[] #);
     enterSpec: @...;
     private: @...;
  enter enterSpec
  #);

(* Portable communication address constructor
 * ==========================================
 *
 * Function. Takes a text value, which is expected to have
 * been produced by some instance X of a specialization of
 * portableCommAddress using its 'asText'. Returns an object
 * with the same value as X.
 *
 * Problems are reported by invoking 'parseError'. The
 * application will then terminate with an exception,
 * unless you extend parseError to leave it.
 *)
portableCommAddressFromText:
  (#
     parseError:<
       (# msg: ^text;
       enter msg[]
       ...
       #);
     txt: ^text;
     addr: ^portableCommAddress;
     <<SLOT portableCommAddressFromTextLib:attributes>>;
  enter txt[]
  ...
  exit addr[]
  #);

(* Portable multicast address
 * ==========================
 *
 * Specifies identities of the members of a group of
 * communication destinations.
 *
 * The group can be built from scratch or enhanced
 * by 'insert'ing members. It can be reduced by
 * 'delete'ing members.
 *)
portableMultiAddress: portableCommAddress
  (#
     init::< (# ... #);

     insert:
       (# addr: ^portablePortAddress;
       enter addr[]
       ...
       #);
```

```
      delete:
        (# addr: ^portablePortAddress;
        enter addr[]
        ...
        #);

      (* private *)
      asTextPattern::< (# ... #);
      private2: @...;
  #);

(* Portable communication port address
 * ===================================
 *
 * Specifies identity of one logical communication destination.
 * A logical destination corresponds to a number of concrete
 * communication ports, represented by instances of
 * specializations of concretePortAddress.
 *
 * A portablePortAddress can be built from scratch by
 * by 'insert'ing such instances. Only one concrete address
 * is allowed for each known type - inserting a second instance
 * overrides the previously inserted one.
 *)
portablePortAddress: portableCommAddress
  (#
      insert:
        (# addr: ^concretePortAddress;
           addrHasUnknownType:< exception;
        enter addr[]
        ...
        #);
      delete:
        (# prot: @integer; (* one of 'commProtocol_.*' *)
           addrHasUnknownType:< exception;
        enter prot
        ...
        #);
      getTcpPort:
        (# addr: ^tcpPortAddress;
        ...
        exit addr[] (* NONE if not present *)
        #);
      getUdpPort:
        (# addr: ^udpPortAddress;
        ...
        exit addr[] (* NONE if not present *)
        #);
      getUnixPort:
        (# addr: ^unixPortAddress;
        ...
        exit addr[] (* NONE if not present *)
        #);
      getPpcPort:
        (# addr: ^ppcPortAddress;
        ...
        exit addr[] (* NONE if not present *)
        #);
      getMemPort:
        (# addr: ^memPortAddress;
        ...
        exit addr[] (* NONE if not present *)
        #);

      (* private *)
      asTextPattern::< (# ... #);
```

```
    private2: @...;
  #);

(* Concrete communication port address
 * ===================================
 *
 * Abstract superpattern for specifying the address
 * of a concrete communication port, such as a UN*X
 * stream socket, a Mac PPC ToolBox session, a shared
 * memory buffer etc.
 *
 * Is able to express its value textually with 'asText',
 * and to characterize its communication protocol
 * with 'commType'.
 *)
concretePortAddress:
  (#
     asText: @asTextPattern;
     asTextPattern:< (# t: ^text do INNER exit t[] #);

     protocol:< integerValue; (* one of 'commProtocol_.*' *)
     protName:< (# t: ^text do &text[] -> t[]; INNER exit t[] #);
     conformsTo: BooleanValue
       (# p: @integer;
       enter p
       ...
       #);
     private: @...;
  #);

(* Unix communication port address types
 * =====================================
 *
 * The pattern unixAbstractPortAddress captures similarities
 * between TCP and UDP ports, represented by
 * tcpPortAddress and udpPortAddress.
 *
 * The pattern unixPortAddress represents an AF_UNIX address
 * family socket, i.e. it appears as a name in some directory,
 * just like a file.
 *
 * NB: The tcpPortAddress also fits a MacTCP port.
 *)
unixAbstractPortAddress: concretePortAddress
  (#
     inetAddr: @integer;
     portNo: @integer;
     asTextPattern::< (# ... #);
  #);

tcpPortAddress: unixAbstractPortAddress
  (#
     protocol::< (# do commProtocol_tcp -> value #);
     protName::< (# do commProtName_tcp -> t #);
  #);

udpPortAddress: unixAbstractPortAddress
  (#
     protocol::< (# do commProtocol_udp -> value #);
     protName::< (# do commProtName_udp -> t #);
  #);

unixPortAddress: concretePortAddress
  (#
     asTextPattern::< (# ... #);
     pathName: @text;
```

```
    protocol::< (# do commProtocol_unix -> value #);
    protName::< (# do commProtName_unix -> t #);
  #);

(* Mac communication port address
 * ==============================
 *
 * Represents a PPC ToolBox session.
 *)
ppcPortAddress: concretePortAddress
  (#
    host: @text;
    portNo: @integer;
    sessionId: @integer;
    asTextPattern::< (# ... #);
    protocol::< (# do commProtocol_ppc -> value #);
    protName::< (# do commProtName_ppc -> t #);
  #);

(* Shared memory buffer port address
 * =================================
 *
 * Corresponding communication support NOT IMPLEMENTED.
 * Could be very fast, perhaps for communicating within
 * one process, using the same source code as for remote
 * communication.
 *)
memPortAddress: concretePortAddress
  (#
    bufferID: @integer; (* !!! This may have to change *)
    asTextPattern::< (# ... #);
    protocol::< (# do commProtocol_mem -> value #);
    protName::< (# do commProtName_mem -> t #);
  #);
```

9.2 commError

```
(* Communication error messages
 * ============================
 *
 * !!! These are obsolete - to be removed.
 *
 * May be returned from communication operations on a
 * connectionPool. Reports OS level errors related to the
 * individual connection in the connectionPool used for
 * the transfer.
 *)
commError_noError:         (# exit  0 #);
commError_noHost:          (# exit -1 #);
commError_connRefused:     (# exit -2 #);
commError_timeOut:         (# exit -5 #);
commError_connBroken:      (# exit -6 #);
commError_nomoreSockets:   (# exit -8 #);

(* Reliability
 * ===========
 *
 * Used to specify the reliability properties
 * required for a transfer (in a protocolSpec).
 * The proporties are additive.
 *)
```

```
commRely_dontcare:   (# exit 0 #);
commRely_loss:       (# exit 2 #); (* packets are not lost *)
commRely_dup:        (# exit 4 #); (* packets are not duplicated *)
commRely_order:      (# exit 8 #); (* packets arrive in correct order
*)
commRely_contents:   (# exit 16 #); (* corrupt data unlikely (e.g.
checksum) *)

commRely_unreliable: (# exit 1 #); (* ensures none of the above *)
commRely_reliable:   (# exit 31 #); (* ensure loss, dup, order, contents *)

(* Type of connection protocol
 * ==========================
 *
 * OS level category of connection. An implementation
 * level description of an individual connection
 * managed by a connectionPool. Weird numbers chosen
 * to make data containing these constants recognizable
 * in a raw communication dump.
 *)
commProtocol_dontcare:   (# exit 0 #);
commProtocol_tcp:        (# exit 72301 #); (* TCP/IP *)
commProtocol_udp:        (# exit 72302 #); (* UDP/IP *)
commProtocol_unix:       (# exit 72303 #); (* UNIX domain (socket as
file) *)
commProtocol_ppc:        (# exit 72304 #); (* Mac PPC ToolBox *)
commProtocol_mem:        (# exit 72305 #); (* Shared memory buffer *)

(* Mnemonic names of the protocols *)
commProtName_tcp:        (# exit 'TCP' #);
commProtName_udp:        (# exit 'UDP' #);
commProtName_unix:       (# exit 'UNIX' #);
commProtName_ppc:        (# exit 'PPC' #);
commProtName_mem:        (# exit 'MEM' #);
```

9.3 communication

```
(* Communication concepts:
 * Pipe: Communication channel between two processes.
 *       For pure standard communication, using standard input/output.
 *       Both processes are unaware of the identity of their
 *       communication partner.
 *
 * Socket: A stream, conceptually an endpoint of a two-way
 *         comminication line. Two endpoints are connected by letting
 *         an ActiveSocket connect to aPassiveSocket. The
 *         PassiveSocket just waits for the ActiveSocket to connect.
 *         After connection both sockets can read/write on
 *         theirstreams.
 *
 * SocketGenerators are used in client/server type communication.
 * Sockets are divided into the categories stream socket and binary
 * socket.
 *
 * Stream sockets:
 *
 * A stream socket is suitable for transferring data, which is
 * readable for human beings, like the data transferred in a UNIX
 * 'talk' session, or like the more formal communication between a
 * smail program and an SMTP mail erver. A stream socket is a stream,
 * so you may 'put', 'get' etc.
```

```
* However, don't rely on this kind of socket when transferring data
* which may contain zero-valued bytes, such as arbitrary binary data.
*
* Binary sockets:
*
* A binary socket is guaranteed to transfer any given block of
* arbitrary bytes unmodified, but you must always specify the
* length of the data block, both for sending and receiving. You may
* 'readData' and 'writeData' on a binary socket, which constitutes
* the lowest level interface.
*
* The operations 'getBlock' and 'putBlock' provide support for
* a very simple, binary data transfer protocol. In this protocol,
* all data is transferred in blocks with the following layout:
*
*      len      header   data
*      |--------|--------|---------------------------------|
*
* The 'len' field is a four byte integer value, in big-endian byte
* order. The 'header' field is a four byte big-endian integer value,
* identifying the kind of data in the 'data' field, the purpose
* of the block, or whatever. The 'data' field length is 4*'len'
* bytes. The sender and the recipient must agree on the
* interpretation of the 'header' and 'data' fields, which is left
* unspecified by this protocol.
*
* The operations 'putRep' and 'getRep' are provided for transferring
* data to and from a ExtendedRepstream object, using this protocol.
* The usage of this level of functionality is recommended whenever
* possible, as it encapsulates (and hides) references to raw memory
* addresses.
*
* The 'Idle' patterns:
*
* Many operations on sockets have an 'Idle' virtual pattern.
* It may be executed one or more times if the operation cannot
* finish right away. This is not guaranteed to happen, so don't
* rely on 'Idle' being executed even once. Extend this virtual
* to keep your application "alive" during a (possibly) lenghty
* operation. Don't execute operations on this(Socket) in an
* enclosed 'Idle'. Don't stop the operation from within an 'Idle' -
* the operation is unfinished; you may for instance have received
* half a block, which makes the stop a serious break wrt the
* protocol; use 'nonBlockingScope' and 'Blocking' for this purpose.
*
* The 'nonBlockingScope' and 'Blocking' patterns:
*
* The 'nonBlockingScope' pattern is used for specifying non-blocking
* communication. This means that operations which cannot begin
* right away are discontinued. An example is: We try to read from a
* socket, but no data at all is available to read. If any
* irreversible actions have been taken in an operation (e.g. reading
* a few bytes), it will not be interrupted by the 'nonBlockingScope'
* mechanism. This means it is always safe to interrupt an operation
* by enclosing it in a 'nonBlockingScope', and to retry it later.
*
* With each 'Idle' pattern comes a 'Blocking' virtual. This is
* executed if the current operation is blocking, i.e. if nothing can
* be done right away. You may extend this virtual to take some action
* in response to the operation being blocked. If the operation is
* enclosed in a 'nonBlockingScope', your 'Blocking'-code gets
* executed immediately before the operation is interrupted. If you
* don't want to interruptthe operation, execute 'continue' in the
* extending of 'Blocking'.
*
* USAGE: Normally the communication will be blocking. But if you
```

```
 * enclose an operation in a specialization of 'nonBlockingScope', we
 * 'leave' the 'nonBlockingScope' at the first blocking condition.
 * PLEASE NOTE: it is unsafe to execute
 * a 'leave' statement which leaves a 'nonBlockingScope'. If you
 * need to leave it, execute 'leaveNBScope'. The normal usage with
 * and without 'nonBlockingScope' looks like this:
 *
 *   /* BLOCKING STYLE */
 *   myStreamSocket.getLine  /* waits until data has arrived */
 *    -> reactOnInput;       /* always executed */
 *   reactSomeMore;          /* always executed */
 *   doOtherThings;
 *
 *   /* NONBLOCKING STYLE */
 *   myStreamSocket.nonBlockingScope
 *      (#
 *      do
 *         myStreamSocket.getLine  if no data: leave scope at once
 *          -> reactOnInput;       only executed if data available
 *         reactSomeMore;          only executed if data available
 *      #);
 *   doOtherThings;
 *
 * With some patterns, it is not possible to have a virtual 'Blocking'
 * or 'Idle' pattern. This is because an enter parameter for the
 * operation is supposedly the address of a beta object. Having taken
 * this, it is unsafe to create objects during the execution of the
 * operation. An example is 'BinarySocket.writeData'. However,
 * enclosing such operations in a 'nonBlockingScope' does cause the
 * operation to behave in a non-blocking manner.
 *)

waitForIO:
  (* Make the process sleep until input/output is available,
   * but at most maxWait seconds. If zero is entered (or the enter
   * part isn't evaluated), wait for I/O without timeout.
   *)
  (# maxWait: @integer;
  enter maxWait
  do ...
  #);

assignGuard: (# assigned: @Boolean do true -> assigned #);

propagateException: (# msg: ^Text enter msg[] do INNER #);

pipe:
  (* The pipe is a composition of two interconnected one way streams.
   * What is written on 'writeEnd' can subsequently be read
   * from 'readEnd'.
   *)
  (#
     (* operations *)
     init:<(# error:< propagateException(# do INNER; … #);
       do ...;
       #);

     (* exceptions *)
     pipeException: Exception
       (#
       enter msg
       do (if msg.empty//false then msg.newline if);
          INNER;
       #);
     pipeError:< PipeException;
```

```
      (* attributes *)
      readEnd: ^Stream;
      writeEnd: ^Stream;

      (* private *)
      private: @...;
   #); (* pipe *)

StreamSocket: Stream
   (#
      (* basics *)
      withPE:
        (# error:< propagateException(# do INNER; msg->otherError #);
        do INNER
        #);
      BasicBlocking:
        (# continue: (# do true->doContinue #);
           doContinue: @boolean;
           doIdle:< Object;
        do INNER;
           (if doContinue//false then leaveNBScope if);
           doIdle;
        #);
      Idle:< Object; (* every local 'Idle' executes this global one *)

      (* operations *)
      open: withPE
        (# Idle:< (# do INNER; this(StreamSocket).Idle #);
           Blocking:< BasicBlocking(# doIdle::< (# do … #) do INNER #);
        do ...
        #);
      close:< withPE
        (#
        do ...
        #);
      flush:< withPE
        (# Idle:< (# do INNER; this(StreamSocket).Idle #);
           Blocking:< BasicBlocking(# doIdle::< (# do … #) do INNER #);
        do ...
        #);
      put::<(# Idle:< (# do INNER; this(StreamSocket).Idle #);
           Blocking:< BasicBlocking(# doIdle::< (# do … #) do INNER #);
        do ...
        #);
      get::<(# Idle:< (# do INNER; this(StreamSocket).Idle #);
           Blocking:< BasicBlocking(# doIdle::< (# do … #) do INNER #);
        do ...
        #);
      peek::<(# Idle:< (# do INNER; this(StreamSocket).Idle #);
           Blocking:< BasicBlocking(# doIdle::< (# do … #) do INNER #);
        do ...
        #);
      eos::<(# Idle:< (# do INNER; this(StreamSocket).Idle #);
           Blocking:< BasicBlocking(# doIdle::< (# do … #) do INNER #);
        do ...
        #);
      putText::<(# Idle:< (# do INNER; this(StreamSocket).Idle #);
           Blocking:< BasicBlocking(# doIdle::< (# do … #) do INNER #);
        do ...
        #);
      getLine::<(# Idle:< (# do INNER; this(StreamSocket).Idle #);
           Blocking:< BasicBlocking(# doIdle::< (# do … #) do INNER #);
        do ...
        #);
      getAtom::<(# Idle:< (# do INNER; this(StreamSocket).Idle #);
           Blocking:< BasicBlocking(# doIdle::< (# do … #) do INNER #);
```

```
          do ...
          #);

        (* nonBlockingScope support *)
        (* don't 'leave' a 'nonBlockingScope'. Use 'leaveNBScope' *)
        nonBlockingScope: (# do ... #);
        leaveNBScope: (# do ... #);

        (* exceptions *)
        sSocketException: streamException
          (#
          enter msg
          do (if msg.empty//false then msg.newline if); INNER
          #);
        otherError::< sSocketException;

        (* attributes *)
        port: @assignGuard(# rep: @integer enter rep exit rep #);

        (* private *)
        private: @...;
     #); (* StreamSocket *)

   BinarySocket:
     (#
        (* basics *)
        withPE:
          (# error:< propagateException(# do INNER; msg->otherError #);
          do INNER
          #);
        withIdle: withPE
          (# Idle:< (# do INNER; this(BinarySocket).Idle #);
             Blocking:<(# continue: (# do true->doContinue #);
                  doContinue: @boolean;
               do INNER;
                  (if doContinue//false then leaveNBScope if);
                  Idle;
               #);
          do INNER
          #);
        rawIO: withPE
          (* Abstract pattern. Read/write exactly 'length' bytes of
           * arbitrary data to/from the memory location 'address'.
           * Non-abstract SPECIALIZATIONS MUST BE STATIC items to
           * prevent garbage collection between calculation of 'address'
           * and reference through 'address'.
           *)
          (# address,length: @integer;
          enter (address,length)
          do INNER
          #);
        repIO: withIdle
          (* Abstract pattern. Read/write a block to/from 'rep',
           * returning/using 'header'. The length of the block is
           * stored in/retrived from 'rep.end'.
           *)
          (# rep: ^ExtendedRepstream;
             header: @integer;
          enter rep[]
          do INNER
          #);
        Idle:< Object; (* every local 'Idle' executes this global one *)

        (* operations *)
        open: withIdle(# do ... #);
        close:< withIdle(# do ... #);
```

```
writeData: @rawIO(# do ... #);
readData: @rawIO(# do ... #);
endOfData:
  (* Returns true if no data is immediately available
   * for reading *)
  (# value: @boolean;
  do ...
  exit value
  #);
putBlock: @withPE
  (# length,header,address: @integer;
  enter (length,header,address)
  do ...
  #);
getBlock: @withPE
  (* The 'maxlen' enter parameter specifies the maximum allowed
   * length of the 'data' field in the block. If the block is
   * bigger than that, the rest of 'data' is discarded. The
   * 'length' exit parameter always specifies the block length,
   * so such an overflow has occurred if maxlen<length. If this
   * behaviour is not acceptable, use 'getBlockLen' and
   * 'getBlockRest'.
   *)
  (# address,maxlen,length,header: @integer;
  enter (address,maxlen)
  do ...
  exit (length,header)
  #);
getBlockLen: withIdle
  (* Exits the length of the next block to receive. Make sure
   * the necessary space is available, and then use
   * 'getBlockRest' to read the block.
   *)
  (# length: @integer;
  do ...
  exit length
  #);
getBlockRest: @withPE
  (* Reads the next block. IMPORTANT: assumes the
   * length has been read with 'getBlockLen' as the last
   * operation on this(BinarySocket).
   *)
  (# address,header: @integer;
  enter address
  do ...
  exit header
  #);

putRep: repIO
  (* Read to ExtendedRepstream using
   * above mentioned binary protocol
   *)
  (#
  enter header
  do ...
  #);
getRep: repIO
  (* Write ExtendedRepstream contents
   * using above mentioned binary protocol
   *)
  (#
  do ...
  exit header
  #);

(* nonBlockingScope support *)
```

```
      (* don't 'leave' a 'nonBlockingScope'. Use 'leaveNBScope'. *)
      nonBlockingScope: (# do ... #);
      leaveNBScope: (# do ... #);

      (* exceptions *)
      bSocketException: Exception
        (#
        enter msg
        do (if msg.empty//false then msg.newline if); INNER
        #);
      otherError:< bSocketException;

      (* attributes *)
      port: @assignGuard(# rep: @integer enter rep exit rep #);

      (* private *)
      private: @...;
   #); (* BinarySocket *)

ActiveStreamSocket: StreamSocket
   (* Initiator of socket communication. Initialize 'host' and 'port'
    * and 'connect' to a passive socket to establish communication.
    *)
   (#
      (* operations *)
      connect: open
        (# enter (host,port)
        do ...;
        #);

      (* attributes *)
      host: @assignGuard(# t: @text; enter t exit t #);
   #); (* ActiveStreamSocket *)

ActiveBinarySocket: BinarySocket
   (* Initiator of socket communication. Initialize 'host' and 'port'
    * and 'connect' to a passive socket to establish communication.
    *)
   (#
      (* operations *)
      connect: open
        (# enter (host,port)
        do ...;
        #);

      (* attributes *)
      host: @assignGuard(# t: @text; enter t exit t #);
   #); (* ActiveBinarySocket *)

PassiveStreamSocket: StreamSocket
   (* 'bind' to port and 'awaitConnection'. Other executions can then
    * connect to the port and communicate through the passive socket.
    * Use a 'nonBlockingScope' to interrupt 'awaitConnection', if no
    * connections are being requested.
    *)
   (#
      (* operations *)
      bind:
        (# error:< propagateException(# do INNER; msg->otherError #);
        enter port
        do ...;
        #);
      awaitConnection: open(# do ...; #);
      close::< (# do ... #);

      (* private *)
```

```
    private2: @...;
  #); (* PassiveStreamSocket *)

PassiveBinarySocket: BinarySocket
  (* 'bind' to port and 'awaitConnection'. Other executions can then
   * connect to the port and communicate through the passive socket.
   * Use a 'nonBlockingScope' to interrupt 'awaitConnection', if no
   * connections are being requested.
   *)
  (#
     (* operations *)
     bind:
       (# error:< propagateException(# do INNER; msg->otherError #);
       enter port
       do ...;
       #);
     awaitConnection: open(# do ...; #);
     close::< (# do ... #);

     (* private *)
     private2: @...;
  #); (* PassiveBinarySocket *)

SocketGenerator:
  (* Supports creating multiple connections on a single port number;
   * typically used in an application acting as a server for a number
   * of clients. do 'portNumber -> bind' and use "get???Connection"
   * to establish connections to the clients. Use a 'nonBlockingScope'
   * to avoid waiting if no clients are requesting a connection.
   *
   * "get???Connection" exits a reference to a "???Socket" associated
   * with the new connection. You may use this like:
   *
   *   mySocketGenerator.getStreamConnection -> aStreamSocketRef[];
   *
   * If you want to work with a specialization of the basic socket
   * patterns, extend the virtuals 'streamSocketType' and/or
   * 'binarySocketType'.
   *)
  (#
     (* basics *)
     streamSocketType:< streamSocket;
     binarySocketType:< binarySocket;
     withIdleAndPE:
       (# Idle:< (# do INNER; this(socketGenerator).Idle #);
          Blocking:<(# continue: (# do true->doContinue #);
               doContinue: @boolean;
            do INNER;
               (if doContinue//false then leaveNBScope if);
               Idle;
            #);
          error:< propagateException(# do INNER; msg->otherError #);
       do INNER
       #);
     Idle:< Object; (* every local 'Idle' executes this global one *)

     (* operations *)
     bind: withIdleAndPE
       (#
       enter port
       do ...
       #);
     close: withIdleAndPE
       (#
       do ...
       #);
```

```
    getStreamConnection: withIdleAndPE
      (# sock: ^streamSocketType;
      do ...;
      exit sock[]
      #);
    getBinaryConnection: withIdleAndPE
      (# sock: ^binarySocketType;
      do ...;
      exit sock[]
      #);

    (* nonBlockingScope support *)
    (* don't 'leave' a 'nonBlockingScope'. Use 'leaveNBScope'. *)
    nonBlockingScope: (# do ... #);
    leaveNBScope: (# do ... #);

    (* exceptions *)
    socketGeneratorException: Exception
      (#
      enter msg
      do (if msg.empty//false then msg.newline if); INNER
      #);
    otherError:< socketGeneratorException;

    (* attributes *)
    port: @assignGuard(# rep: @integer enter rep exit rep #);

    (* private *)
    private: @...;
 #); (* SocketGenerator *)
```

9.4 connectionPool

```
(* A connectionPool manages a number of client side
 * communication interfaces (e.g. active sockets), and
 * allows choosing which one of those to use for a
 * communication transfer by means of a
 * portableCommAddress.
 *
 * The communication interfaces are subject to concurrency
 * control, so they must be used in a 'take-it, use-it,
 * give-it-back' fashion. This is achieved by the pattern
 * 'communication' in 'connectionPool'.
 *)

(* The binary connection pool
 *
 * Instances of BinaryConnectionPool are used for managing
 * a number of binary socket connections. The user of a
 * BinaryConnectionPool gives a specification of the
 * receiver, the type of connection, the quality of
 * service etc. in a portableCommAddress to a (specialization
 * of) the control pattern 'communication'. This is used as
 * follows (bcPool is an instance of BinaryConnectionPool):
 *
 *   addr[] -> bcPool.communication
 *     (# Extend error callbacks here
 *     do
 *        Within this dopart: use 'sock' to communicate
 *        When leaving, forget 'sock' (don`t bring out ref.s to it)
 *     #);
```

```
 *
 * If you want to 'leave' the dopart of a specialization of
 * a 'communication', use
 *
 *   leaving(# do leave L #);
 *
 * in stead of
 *
 *   leave L;
 *
 * Otherwise some resources may be rendered inaccessible.
 *)
BinaryConnectionPool:
  (#
     (* patterns *)
     socketType:< activeBinarySocket;

     (* operations *)
     init:<
       (#
       do ...
       #);
     communication:
       (# addr: ^portableCommAddress;
          sock: ^socketType;
          leaving: (# do ... #);

          (* hooks *)
          onNewConnection:<
            (* executed when a new connection has been created *)
            (# sock: ^socketType; (* The new connection *)
               context: ^object; (* NB: Should`ve been private *)
               actor: ^|system; (* process to associate with sock *)
            enter (sock[],context[])
            do INNER
            exit actor[]
            #);

          (* operations *)
          removeSock: (* remove sock from this pool *)
            (# dopart: @...;
            do dopart
            #);

          (* exceptions *)
          error:< hiErrCB (* operation level error callback *)
            (#
            do INNER;
               …
            #);
          concrErrCB: hiErrCB
             (*superpattern for concrete error callbacks*)
            (#
            do INNER;
               …
            #);
          addrHasUnknownType:< exception;
             (* Considered fatal, for now *)
          internalError:< concrErrCB(# do INNER #);
          unknownError:< concrErrCB(# do INNER #);
          accessError:< concrErrCB(# do INNER #);
          resourceError:< concrErrCB(# do INNER #);
          addressError:< concrErrCB(# do INNER #);
          refusedError:< concrErrCB(# do INNER #);
          intrError:< concrErrCB(# do INNER #);
          getHostError:< concrErrCB(# do INNER #);
```

```
        (* private *)
        priv: @...;

      enter addr[]
      do ...
      #);
    markAsDead:
      (# dopart: @...;
         sock: ^binarySocket;
      enter sock[]
      do dopart
      #);
    removeSomeConnection:
      (* Removes least recently used currently unused connection *)
      (# noConnectionsRemovable:< object;
         dopart: @...;
      do dopart
      #);
    close:<
      (#
      do ...
      #);

    (* top level error callback *)
    error:< hiErrCB(# do INNER #);

    (* private *)
    private: @...;
  #);
```

9.5 errorCallback

```
(* Basic Exception Handling
 * ========================
 *
 * Whenever an error condition is detected on a socket, a
 * corresponding virtual pattern is instantiated and executed.
 * These patterns are specializations of 'errCB', as
 * declared below. Such virtual patterns are hereafter denoted
 * error callback patterns. To catch and treat an error,
 * extend the corresponding error callback.
 *
 * If an error callback is not extended and the
 * corresponding error occurs, an exception is executed
 * and the program terminates. If the error callback
 * is extended, the following holds:
 *
 *   - if 'abort' is executed in the extending dopart,
 * the operation (but not the program) is aborted. You may
 * execute 'leave' within a specialization of abort. Don't
 * 'leave' an error callback from any other point, as this
 * may put the object or the process into an unstable
 * state. If you 'abort' but do not 'leave', the operation
 * aborts, but control flow is like when the operation succeeds;
 * in this case, any exited values are dummy values, reflecting
 * that the operation failed. Don't use them!
 *
 *   - if 'continue' is executed in the extending dopart,
 * there will be an attempt to recover and finish the operation,
 * after the execution of the error callback terminates.
```

```
 *
 *   - if 'fatal' is executed in the extending dopart,
 * an exception will be executed and the program terminated,
 * before the execution of the error callback returns. (This
 * is also the default, but with hierarchical error callbacks,
 * you may need 'fatal' to undo a 'continue' at a higher level).
 *
 * In case it happens more than once that an operation
 * from the set 'abort','continue','fatal' is executed,
 * the one executed as the last takes precedence.
 *
 * Propagating exceptions
 * ======================
 *
 * The error callback patterns are present at three different
 * levels: Concrete error callbacks, operation level error
 * callbacks, and socket level error callbacks.
 *
 * The concrete error callbacks provide the greatest level of
 * detail: their names indicate the kind of error condition
 * detected. This makes it possible to treat different errors
 * differently.
 *
 * The operation level error callback is executed whenever
 * an error condition is detected during the execution of
 * that operation. In a extending of this kind of error
 * callback, you can adjust the default action for all the
 * concrete error callbacks in this operation.
 *
 * The single socket level error callback is executed whenever
 * any operation detects any error condition. In a extending
 * of this error callback, you can adjust the default action
 * for all operation level error callbacks.
 *
 * The means for adjusting the behaviour is in all cases to
 * execute 'abort' (probably "abort(# leave L #)") 'continue',
 * or 'fatal', and the semantics of these imperatives are
 * like in concrete error callbacks.
 *
 * Error callback extendings normally take precedence
 * like this: concrete > operation level > socket level.
 * This means that the higher level specifies a default, and
 * the more concrete level overrides this default if it
 * executes 'continue', 'abort', or 'fatal'. This doesn't
 * hold, however, if you "abort(# do leave L #)" at a higher
 * level: In this case, the more concrete levels will never
 * get a chance to undo the 'leave'.
 *)
--- lib:attributes ---

errCB_initialValue: (# exit -1 #);
errCB_abortProgram: (# exit 0 #);
errCB_abortOperation: (# exit 1 #);
errCB_continueOperation: (# exit 2 #);

errCB: IntegerValue
  (# abort: (# do ... #);
     continue: (# do ... #);
     fatal: (# do ... #);
     addMsg: (# t: ^text enter t[] ... #);
     exceptionType:< exception;
     cleanup: ^object;
     private: @...;
  enter cleanup[]
  do ...
  #);
```

```
hiErrCB: IntegerObject
  (# abort: (# do ... #);
     continue: (# do ... #);
     fatal: (# do ... #);
     cleanup: ^object;
  enter cleanup[]
  do INNER
  #);
```

9.6 idScheduler

```
idSchedElement:
  (#
     suspend_sem: @semaphore;
     id: @integer;
  #);

idScheduler:
  (#
     isElement:< idSchedElement;
     prefix:
       (# id: @integer;
          elm: ^isElement;
       enter id
       do INNER
       #);

     (* operations *)
     init:<
       (#
       do ...
       #);
     id_suspend: prefix
       (# dopart: @...;
       do dopart; INNER;
       #);
     id_resume: prefix
       (# found:< object;
          not_found:< object;
          dopart: @...;
       do dopart
       #);

     (* private *)
     private: @...
  #);

idTimeoutScheduler: idScheduler
  (#
     (* operations *)
     init::<
       (#
       do ...
       #);
     id_timeoutSuspend: prefix
       (# timeoutValue: @integer;
          retry:< BooleanValue;
          onSuccess:< object;
          onTimeout:< object;
          dopart: @...;
```

```
      enter timeoutValue
      do dopart
      #);

    (* private *)
    private2: @...;
  #);
```

9.7 osinterface

```
osinterface:
  (#
    <<SLOT OSInterfaceLib:attributes>>;

    init:< (# do ...; INNER  #);
    hostMachine:
      (# theMachine: @text
      do ...
      exit theMachine
      #);
    hostName:
      (# error:<OSError;
         result: @text;
      do ...;
      exit result
      #);
    getHostAddr:
      (# addr: ^Text;
      do ...
       exit addr[]
       #);
    thisProcess: @Process
      (#
         scanArguments:
           (# current: @Text;
           do ...;
           #);
      #); (* the process referring to this program execution *)

  do init; INNER;
  #);

OSError: Exception
  (# message: @Text;
  enter message
  do ...;
     INNER;
  #);
```

9.8 processmanager

```
(* The ProcessManager models the concepts of program executions and
 * communication between program executions.
 *
 * A program execution is modelled as a process.
 * A process can be started (executed) and stopped (terminated).
```

```
 * Processes can communicate to each other using either pipes or
 * sockets. Using pipes as communication model, processes can make
 * simple communication though redirection of standard input/output
 * streams (screen and keyboard). Using sockets as communication
 * model, processes can communicate through any specified stream.
 *)

(* Notice, this(Process) can only be executed once.
 *
 * Two program executions of the same Process,
 * can be executed by instantiating and executing two different BETA
 * objects from the same Process.
 *)

Process:
  (#
     <<SLOT ProcessLib:attributes>>;

     name: ^Text;
     init:< (# enter name[] do ...; INNER #);

     argType:
       (# argument: @Text;
          putArg:
            (# t: ^Text;
            enter t[]
            do ...
            #);
          append: @putArg;
          scanArguments: (* calls INNER for each argument *)
            (# current: @Text;
            do ...;
            #);
       #);
     argument: @argType; (* arguments to this(Process) *)

       (* operations *)

     start: (* starts this(Process)'s program execution *)
       (# error:< ProcessManagerException;
          twoCurrent:< ProcessManagerException;
       do ...; INNER;
       #);

     stop: (* stops this(Process)'s program execution *)
       (# error:< ProcessManagerException;
       do ...; INNER;
       #);

     awaitStopped: (* Returns when THIS(Process) stops *)
       (# error:< ProcessManagerException;
       do ...; INNER;
       #);

     stillRunning: (* Returns true if THIS(Process) is still running*)
       (# error:< ProcessManagerException;
          value: @Boolean;
       do ...; INNER;
       exit value
       #);

     (* input/output redirection *)

     connectToProcess:
       (* connect output of this(process) to toProcess's input
        * In Unix terms: this(Process) | toProcess
```

```
        *)
       (# error:< ProcessManagerException;
          toProcess: ^Process;
       enter toProcess[]
       do ...;
       #);

     connectInPipe:
       (* connect output of fromProcess to input of  this(process)
        * In Unix terms: fromProcess | this(Process)
        *)
       (# error:< ProcessManagerException;
          fromProcess: ^Process;
       enter fromProcess[]
       do ...;
       #);

     redirectFromFile:
       (* redirect input to this(process) from inputFile
        * In Unix terms: this(Process) < inputFile
        *)
       (# error:< ProcessManagerException;
          inputFile: ^File;
       enter inputFile[]
       do ...;
       #);

     redirectToFile:
       (* redirect output of this(process) to outputFile
        * In Unix terms: this(Process) > outputFile
        *)
       (# error:< ProcessManagerException;
          outputFile: ^File;
       enter outputFile[]
       do ...;
       #);

     redirectFromChannel:
       (* redirect input to this(process) from inputChannel *)
       (# error:< ProcessManagerException;
          inputChannel: ^Stream;
       enter inputChannel[]
       do ...;
       #);

     redirectToChannel:
       (* redirect output of this(process) to outputChannel *)
       (# error:< ProcessManagerException;
          outputChannel: ^Stream;
       enter outputChannel[]
       do ...;
       #);

   (* Virtual callbacks: called when the proper action has occurred *)

     onStart:< (# do INNER #);
     onStop:< (# do INNER #);

     doDebug: @Boolean;
     private: @...;
  #);

ProcessManagerException: Exception
  (# message: ^Text;
  enter message[]
  do ...;
```

```
    INNER;
 #);
```

9.9 systemComm

```
(* Expected context:
 * =================
 *
 * Instances of the patterns of this fragment are expected to be
 * executed from components (co-routines). Whenever an operation
 * is about to block, the current component will be suspended.
 * It will be resumed some time later, when the requested IO
 * is available. This means that communication related
 * functionality can be written in a simple, blocking style; it
 * will behave approximately as if the scheduler were preemptive.
 *
 * Communication concepts:
 * =======================
 *
 * Pipe: Communication channel between two processes.
 *       For pure standard communication, using standard input/output.
 *       Both processes are unaware of the identity of their
 *       communication partner.
 *
 * Socket: A stream, conceptually an endpoint of a two-way
 *         comminication line. Two endpoints are connected by letting
 *         an ActiveSocket connect to a PassiveSocket. The
 *         PassiveSocket just waits for the ActiveSocket  to connect.
 *         After connection both sockets can read/write on their
 *         streams.
 *
 * SocketGenerators are used in client/server type communication.
 * Sockets are divided into the categories stream socket and binary
 * socket.
 *
 * Stream sockets:
 *
 * A stream socket is suitable for transferring data, which is
 * readable for human beings, like the data transferred in a UNIX
 * 'talk' session, or like the more formal communication between a
 * mail program and an SMTP mail server. A stream socket is a stream,
 * so you may 'put', 'get' etc. However, don't use this kind of socket
 * when transferring data which may contain zero-valued bytes, such as
 * arbitrary binary data.
 *
 * Binary sockets:
 *
 * A binary socket is guaranteed to transfer any given block of
 * arbitrary bytes unmodified, but you must always specify the
 * length of the data block, both for sending and receiving. You may
 * 'readData' and 'writeData' on a binary socket, which constitutes
 * the lowest level interface.
 *
 * The operations 'getBlock' and 'putBlock' provide support for
 * a very simple, binary data transfer protocol. In this protocol,
 * all data is transferred in blocks with the following layout:
 *
 *      len      header   data
 *      |--------|--------|---------------------------------|
 *
 * The 'len' field is a four byte integer value, in big-endian byte
```

```
 * order. The 'header' field is a four byte big-endian integer value,
 * identifying the kind of data in the 'data' field, the purpose
 * of the block, or whatever. The 'data' field length is 4*'len'
 * bytes. The sender and the recipient must agree on the
 * interpretation of the 'header' and 'data' fields, which is left
 * unspecified by this protocol.
 *
 * The operations 'putRep' and 'getRep' are provided for transferring
 * data to and from a ExtendedRepstream object, using this protocol.
 * The usage of this level of functionality is recommended whenever
 * possible, as it encapsulates (and hides) references to raw memory
 * addresses.
 *
 * The operations 'putRepObj' and 'getRepObj' are similar to 'putRep'
 * and 'getRep', apart from: (1) The objects sent or received are
 * instances of the pattern RepetitionObject. (2) the protocol has
 * no header field, and the length field is the first element in
 * the repetition from the repetitionObject:
 *
 *      len      data
 *      |--------|---------------------------------|
 *
 * Otherwise, it is like the above protocol.
 *
 * The 'Idle' patterns:
 * ====================
 *
 * Many operations on sockets have an 'Idle' virtual pattern.
 * It may be executed one or more times if the operation cannot
 * finish right away. This is not guaranteed to happen, so don't
 * rely on 'Idle' being executed even once. Extend this virtual
 * to keep your application "alive" during a (possibly) lenghty
 * operation. Don't execute operations on this(Socket) in an
 * enclosed 'Idle'. Don't stop the operation from within an 'Idle' -
 * the operation is unfinished; you may for instance have received
 * half a block, which makes the stop a serious break wrt the
 * protocol; use 'nonBlockingScope' and 'Blocking' for this purpose.
 *
 * The 'nonBlockingScope' and 'Blocking' patterns:
 * ===============================================
 *
 * The 'nonBlockingScope' pattern is used for specifying non-blocking
 * communication. This means that operations which cannot begin
 * right away are discontinued. An example is: We try to read from a
 * socket, but no data at all is available to read. If any
 * irreversibleactions have been taken in an operation (e.g. reading a
 * few bytes), it will not be interrupted by the 'nonBlockingScope'
 * mechanism. This means it is always safe to interrupt an operation
 * by enclosing it in a 'nonBlockingScope', and to retry it later.
 *
 * With each 'Idle' pattern comes a 'Blocking' virtual. This is
 * executed if the current operation is blocking, i.e. if nothing can
 * be done right away. You may extend this virtual to take some action
 * in response to the operation being blocked. If the operation is
 * enclosed in a 'nonBlockingScope', your 'Blocking'-code gets
 * executed immediately before the operation is interrupted. If you
 * don't want to interrupt the operation, execute 'continue' in the
 * extending of 'Blocking'.
 *
 * USAGE: Normally the communication will be blocking. But if you
 * enclosean operation in a specialization of 'nonBlockingScope', we
 * 'leave' the 'nonBlockingScope' at the first blocking condition.
 * PLEASE NOTE: it is unsafe to execute a 'leave' statement which
 * leaves a 'nonBlockingScope'. If you need to leave it, execute
 * 'leaveNBScope'. The normal usage with and without
 * 'nonBlockingScope' looks like this:
```

```
 *
 *   BLOCKING STYLE
 *   myStreamSocket.getLine  waits until data has arrived
 *    -> reactOnInput;       always executed
 *   reactSomeMore;          always executed
 *   doOtherThings;
 *
 *   NONBLOCKING STYLE
 *   myStreamSocket.nonBlockingScope
 *      (#
 *      do
 *         myStreamSocket.getLine  if no data: leave scope at once
 *          -> reactOnInput;      only executed if data available
 *         reactSomeMore;         only executed if data available
 *      #);
 *   doOtherThings;
 *
 * With some patterns, it is not possible to have a virtual 'Blocking'
 * or 'Idle' pattern. This is because an enter parameter for the
 * operation is supposedly the address of a beta object. Having taken
 * this, it is unsafe to create objects during the execution of the
 * operation. An example is 'BinarySocket.writeData'. However,
 * enclosing such operations in a 'nonBlockingScope' does cause the
 * operation to behave in a non-blocking manner.
 *
 * Exception Handling
 * ==================
 *
 * Uses error callbacks. Read about these in 'errorCallback.bet'.
 *
 * The error callbacks used have the following meaning:
 *
 *
 *   Error callback name      Meaning
 * ----------------------------------------------------
 *   accessError              Insufficient access rights
 *   addressError             Address (i.e. (host,port)) in use or
 *                            invalid
 *   badMsgError              (hardly documented in man page)
 *   connBrokenError          The connection has become unusable
 *   eosError                 End-of-stream
 *   getHostError             Error when getting hostname
 *   internalError            Should not happen; please report if it
 *                            does!
 *   intrError                Operation interrupted by signal
 *   refusedError             Connection refused by peer
 *   resourceError            Too few file descriptors/buffers etc.
 *   timedOut                 Specified timeout period has expired
 *   timedOutInTransfer       Timed out, and some data have been
 *                            transferred
 *   unknownError             OS reports unknown errno (new OS?)
 *   usageError               Eg: you must initialize port before
 *                            connecting
 *
 *   nospaceError             (StreamSocket) returned by op. on
 *                            fdStream
 *   otherError               (StreamSocket) from fdStream
 *   readError                (StreamSocket) from fdStream
 *   writeError               (StreamSocket) from fdStream
 *   accessError              (also occurs as an fdStream error)
 *)

waitForever: (* Default for timeouts *)
  IntegerValue(# do -1->value; INNER #);

assignGuard: (# assigned: @Boolean do true -> assigned #);
```

```
propagateException: (# msg: ^Text enter msg[] do INNER #);

pipe:
  (* The pipe is a composition of two interconnected one way streams.
   * What is written on 'writeEnd' can subsequently be read
   * from 'readEnd'.
   *)
  (#
     (* operations *)
     init:<(# error:< propagateException(# do INNER; … #);
       do ...
       #);

     (* !!!! exceptions *)
     pipeException: Exception
       (#
       enter msg
       do (if msg.empty//false then msg.newline if);
          INNER;
       #);
     pipeError:< PipeException;

     (* attributes *)
     readEnd: ^Stream;
     writeEnd: ^Stream;

     (* private *)
     private: @...;
  #); (* pipe *)

StreamSocket: Stream
  (#
     (* basics *)
     withPE:
       (# error:< hiErrCB (* operation level error callback *)
            (#
            do INNER;
               (if errCB_initialValue // value then
                   (value,cleanup[])->this(StreamSocket).error->value;
               if);
            #);
          loErrCB: errCB (*superpattern for concrete error callbacks*)
            (#
            do INNER;
               (if errCB_initialValue // value then
                   (value,cleanup[])->error->value;
               if);
            #);
          accessError:< loErrCB(# do INNER #);
          nospaceError:< loErrCB(# do INNER #);
          writeError:< loErrCB(# do INNER #);
          usageError:< loErrCB(# do INNER #);
          otherError:< loErrCB(# do INNER #);
          timedOut:< loErrCB(# do INNER #);
          timeout: @integer;
          enter timeout
       do INNER
       #);
     BasicBlocking:
       (# continue: (# do true->doContinue #);
          doContinue: @boolean;
          doIdle:< Object;
       do INNER;
          (if doContinue//false then leaveNBScope if);
          doIdle;
```

```
  #);
Idle:< Object; (* every local 'Idle' executes this global one *)
timeoutValue:< waitForever; (*length in seconds, all operations*)

(* operations *)
sameConnection: booleanValue
  (* do 'this' and 'other' wrap the same OS level connection? *)
  (# other: ^StreamSocket;
  enter other[]
  ...
  #);
getPortableAddress:
  (# addr: ^portablePortAddress;
     dopart: @...;
  do dopart
  exit addr[]
  #);
open: withPE
  (# Idle:< (# do INNER; this(StreamSocket).Idle #);
     Blocking:< BasicBlocking(# doIdle::< (# do … #) do INNER #);
  do ...
  #);
close:< withPE
  (#
  do ...
  #);
flush: withPE
  (# Idle:< (# do INNER; this(StreamSocket).Idle #);
     Blocking:< BasicBlocking(# doIdle::< (# do … #) do INNER #);
     dopart: @...;
  do dopart
  #);
put::
  (# Idle:< (# do INNER; this(StreamSocket).Idle #);
     Blocking:< BasicBlocking(# doIdle::< (# do … #) do INNER #);
     error:< hiErrCB (* operation level error callback *)
       (#
       do INNER;
          …
       #);
     loErrCB: errCB (*superpattern for concrete error callbacks*)
       (#
       do INNER;
          …
       #);
     writeError:< loErrCB(# do INNER #);
     timedOut:< loErrCB(# do INNER #);
     dopart: @...;
  do dopart
  #);
get::
  (# Idle:< (# do INNER; this(StreamSocket).Idle #);
     Blocking:< BasicBlocking(# doIdle::< (# do … #) do INNER #);
     error:< hiErrCB (* operation level error callback *)
       (#
       do INNER;
          …
       #);
     loErrCB: errCB (*superpattern for concrete error callbacks*)
       (#
       do INNER;
          …
       #);
     readError:< loErrCB(# do INNER #);
     eosError:< loErrCB(# do INNER #);
     connBrokenError:< loErrCB(# do INNER #);
```

```
        timedOut:< loErrCB(# do INNER #);
        dopart: @...;
     do dopart
     #);
   peek::
     (# Idle:< (# do INNER; this(StreamSocket).Idle #);
        Blocking:< BasicBlocking(# doIdle::< (# do … #) do INNER #);
        error:< hiErrCB (* operation level error callback *)
          (#
          do INNER;
             …
          #);
        loErrCB: errCB (*superpattern for concrete error callbacks*)
          (#
          do INNER;
             …
          #);
        readError:< loErrCB(# do INNER #);
        eosError:< loErrCB(# do INNER #);
        connBrokenError:< loErrCB(# do INNER #);
        timedOut:< loErrCB(# do INNER #);
        dopart: @...;
     do dopart
     #);
   eos::
     (# Idle:< (# do INNER; this(StreamSocket).Idle #);
        Blocking:< BasicBlocking(# doIdle::< (# do … #) do INNER #);
        error:< hiErrCB (* operation level error callback *)
          (#
          do INNER;
             …
          #);
        loErrCB: errCB (*superpattern for concrete error callbacks*)
          (#
          do INNER;
             …
          #);
        connBrokenError:< loErrCB(# do INNER #);
        internalError:< loErrCB(# do INNER #);
        unknownError:< loErrCB(# do INNER #);
        dopart: @...;
     do dopart
     #);
   putText::
     (# Idle:< (# do INNER; this(StreamSocket).Idle #);
        Blocking:< BasicBlocking(# doIdle::< (# do … #) do INNER #);
        error:< hiErrCB (* operation level error callback *)
          (#
          do INNER;
             …
          #);
        loErrCB: errCB (*superpattern for concrete error callbacks*)
          (#
          do INNER;
             …
          #);
        writeError:< loErrCB(# do INNER #);
        timedOut:< loErrCB(# do INNER #);
        dopart: @...;
     do dopart
     #);
   getLine::
     (# Idle:< (# do INNER; this(StreamSocket).Idle #);
        Blocking:< BasicBlocking(# doIdle::< (# do … #) do INNER #);
        error:< hiErrCB (* operation level error callback *)
          (#
```

```
            do INNER;
               …
            #);
          loErrCB: errCB (*superpattern for concrete error callbacks*)
            (#
            do INNER;
               …
            #);
          readError:< loErrCB(# do INNER #);
          eosError:< loErrCB(# do INNER #);
          connBrokenError:< loErrCB(# do INNER #);
          timedOut:< loErrCB(# do INNER #);
          dopart: @...;
       do dopart
       #);
     getAtom::
       (# Idle:< (# do INNER; this(StreamSocket).Idle #);
          Blocking:< BasicBlocking(# doIdle::< (# do … #) do INNER #);
          error:< hiErrCB (* operation level error callback *)
            (#
            do INNER;
               …
            #);
          loErrCB: errCB (*superpattern for concrete error callbacks*)
            (#
            do INNER;
               …
            #);
          readError:< loErrCB(# do INNER #);
          eosError:< loErrCB(# do INNER #);
          connBrokenError:< loErrCB(# do INNER #);
          timedOut:< loErrCB(# do INNER #);
          dopart: @...;
       do dopart
       #);
     forceTimeout:< (# do ... #);
     usageTimestamp:< integerValue(# ... #);

     (* nonBlockingScope support *)
     (* don`t 'leave' a 'nonBlockingScope'. Use 'leaveNBScope' *)
     nonBlockingScope: (# do ... #);
     leaveNBScope: (# do ... #);

     (* socket level error callback *)
     error:< hiErrCB(# do INNER #);

     (* attributes *)
     host: @assignGuard(# t: @text; enter t exit t #);
     port: @assignGuard(# rep: @integer enter rep exit rep #);
     inetAddr: @assignGuard(# rep: @integer enter rep exit rep #);

     (* private *)
     private: @...;
  #); (* StreamSocket *)

BinarySocket:
  (#
     (* basics *)
     withPE:
       (# error:< hiErrCB (* operation level error callback *)
            (#
            do INNER;
               …
            #);
          loErrCB: errCB (*superpattern for concrete error callbacks*)
            (#
```

```
            do INNER;
               …
            #);
          timedOut:< loErrCB(# do INNER #);
          timedOutInTransfer:< loErrCB(# do INNER #);
          internalError:< loErrCB(# do INNER #);
          connBrokenError:< loErrCB(# do INNER #);
          usageError:< loErrCB(# do INNER #);
          unknownError:< loErrCB(# do INNER #);
          timeout: @integer;
       enter timeout
       do INNER
       #);
     withIdle: withPE
       (# Idle:< (# do INNER; this(BinarySocket).Idle #);
          Blocking:<(# continue: (# do true->doContinue #);
               doContinue: @boolean;
            do INNER;
               (if doContinue//false then leaveNBScope if);
               Idle;
            #);
       do INNER
       #);
     repIO: withIdle
       (* Abstract pattern. Read/write a block to/from 'rep',
        * returning/using 'header'. The length of the block is
        * stored in/retrived from 'rep.end'.
        *)
       (# resourceError:< loErrCB(# do INNER #);
          badMsgError:< loErrCB(# do INNER #);
          rep: ^ExtendedRepstream;
          header: @integer;
       enter rep[]
       do INNER
       #);
     repObjIO: withIdle
       (* Abstract pattern. Read/write a block to/from 'rep',
        * The length of the block is stored in/retrived from
        * 'rep.end'.
        *)
       (# resourceError:< loErrCB(# do INNER #);
          badMsgError:< loErrCB(# do INNER #);
          rep: ^RepetitionObject;
       enter rep[]
       do INNER
       #);
     Idle:< Object; (* every local 'Idle' executes this global one *)

     (* operations *)
     sameConnection: booleanValue
       (* do 'this' and 'other' wrap the same OS level connection? *)
       (# other: ^BinarySocket;
       enter other[]
       ...
       #);
     getPortableAddress:
       (# addr: ^portablePortAddress;
          dopart: @...;
       do dopart
       exit addr[]
       #);
     open: withIdle(# do ... #);
     close:< withIdle(# do ... #);
     endOfData: @endOfDataPattern;
     putRep: @putRepPattern;
     getRep: @getRepPattern;
```

```
putRepObj: @putRepObjPattern;
getRepObj: @getRepObjPattern;
forceTimeout:< (# do ... #);
usageTimestamp:< integerValue(# ... #);

endOfDataPattern:
  (* Returns true if no data is immediately
   * available for reading *)
  (# error:< hiErrCB (* operation level error callback *)
       (#
       do INNER;
          …
       #);
     loErrCB: errCB (*superpattern for concrete error callbacks*)
       (#
       do INNER;
          …
       #);
     connBrokenError:< loErrCB(# do INNER #);
     internalError:< loErrCB(# do INNER #);
     unknownError:< loErrCB(# do INNER #);
     value: @boolean;
     dopart: @...;
  do dopart
  exit value
  #);
putRepPattern: repIO
  (* Read to ExtendedRepstream using
   * above mentioned binary protocol
   *)
  (# dopart: @...;
  enter header
  do dopart
  #);
getRepPattern: repIO
  (* Write ExtendedRepstream contents
   * using above mentioned binary protocol
   *)
  (# dopart: @...;
  do dopart
  exit header
  #);
putRepObjPattern: repObjIO
  (* Read to RepetitionObject, using headerless protocol *)
  (# dopart: @...;
  do dopart
  #);
getRepObjPattern: repObjIO
  (* Write RepetitionObject, using headerless protocol *)
  (# dopart: @...;
  do dopart
  #);

(* nonBlockingScope support *)
(* don`t 'leave' a 'nonBlockingScope'. Use 'leaveNBScope'. *)
nonBlockingScope: (# do ... #);
leaveNBScope: (# do ... #);

(* socket level error callback *)
error:< hiErrCB(# do INNER #);

(* attributes *)
host: @assignGuard(# t: @text; enter t exit t #);
port: @assignGuard(# rep: @integer enter rep exit rep #);
inetAddr: @assignGuard(# rep: @integer enter rep exit rep #);
```

```
     (* private *)
     private: @...;
  #); (* BinarySocket *)

ActiveStreamSocket: StreamSocket
  (* Initiator of socket communication. Initialize 'host' and 'port'
   * and 'connect' to a passive socket to establish communication.
   *)
  (#
     (* operations *)
     connect: open
       (# resourceError:< loErrCB(# do INNER #);
          addressError:< loErrCB(# do INNER #);
          refusedError:< loErrCB(# do INNER #);
          intrError:< loErrCB(# do INNER #);
          getHostError:< loErrCB(# do INNER #);
          connBrokenError:< loErrCB(# do INNER #);
          unknownError:< loErrCB(# do INNER #);
          internalError:< loErrCB(# do INNER #);
          dopart: @...;
       enter (host,port)
       do dopart
       #);
  #); (* ActiveStreamSocket *)

ActiveBinarySocket: BinarySocket
  (* Initiator of socket communication. Initialize 'host' and 'port'
   * and 'connect' to a passive socket to establish communication.
   *)
  (#
     (* operations *)
     connect: open
       (# accessError:< loErrCB(# do INNER #);
          resourceError:< loErrCB(# do INNER #);
          addressError:< loErrCB(# do INNER #);
          refusedError:< loErrCB(# do INNER #);
          intrError:< loErrCB(# do INNER #);
          getHostError:< loErrCB(# do INNER #);
          dopart: @...;
       enter (host,port)
       do dopart
       #);
  #); (* ActiveBinarySocket *)

PassiveStreamSocket: StreamSocket
  (* 'bind' to port and 'awaitConnection'. Other executions can then
   * connect to the port and communicate through the passive socket.
   * Use a 'nonBlockingScope' to interrupt 'awaitConnection', if no
   * connections are being requested.
   *)
  (#
     (* operations *)
     bind:
       (# error:< hiErrCB (* operation level error callback *)
            (#
            do INNER;
               …
            #);
          loErrCB: errCB (*superpattern for concrete error callbacks*)
            (#
            do INNER;
               …
            #);
          connBrokenError:< loErrCB(# do INNER #);
          accessError:< loErrCB(# do INNER #);
          addressError:< loErrCB(# do INNER #);
```

```
            intrError:< loErrCB(# do INNER #);
            resourceError:< loErrCB(# do INNER #);
            internalError:< loErrCB(# do INNER #);
            unknownError:< loErrCB(# do INNER #);
            usageError:< loErrCB(# do INNER #);
            dopart: @...;
         enter port
         do dopart
         #);
       awaitConnection: open
         (# connBrokenError:< loErrCB(# do INNER #);
            resourceError:< loErrCB(# do INNER #);
            internalError:< loErrCB(# do INNER #);
            unknownError:< loErrCB(# do INNER #);
            dopart: @...;
         do dopart
         #);
       close::< (# do ... #);
       forceTimeout::< (# do ... #);
       usageTimestamp::< (# ... #);

       (* private *)
       private2: @...;
    #); (* PassiveStreamSocket *)

  PassiveBinarySocket: BinarySocket
    (* 'bind' to port and 'awaitConnection'. Other executions can then
     * connect to the port and communicate through the passive socket.
     * Use a 'nonBlockingScope' to interrupt 'awaitConnection', if no
     * connections are being requested.
     *)
    (#
       (* operations *)
       bind:
         (# error:< hiErrCB (* operation level error callback *)
              (#
              do INNER;
                 …
              #);
            loErrCB: errCB (*superpattern for concrete error callbacks*)
              (#
              do INNER;
                 …
              #);
            connBrokenError:< loErrCB(# do INNER #);
            accessError:< loErrCB(# do INNER #);
            addressError:< loErrCB(# do INNER #);
            intrError:< loErrCB(# do INNER #);
            resourceError:< loErrCB(# do INNER #);
            internalError:< loErrCB(# do INNER #);
            unknownError:< loErrCB(# do INNER #);
            usageError:< loErrCB(# do INNER #);
            dopart: @...;
         enter port
         do dopart
         #);
       awaitConnection: open
         (# accessError:< loErrCB(# do INNER #);
            connBrokenError:< loErrCB(# do INNER #);
            resourceError:< loErrCB(# do INNER #);
            dopart: @...;
         do dopart
         #);
       close::< (# do ... #);
       forceTimeout::< (# do ... #);
       usageTimestamp::< (# ... #);
```

```
    (* private *)
    private2: @...;
  #); (* PassiveBinarySocket *)

SocketGenerator:
  (* Supports creating multiple connections on a single port number;
   * typically used in an application acting as a server for a number
   * of clients. do 'portNumber -> bind' and use "get???Connection"
   * to establish connections to the clients. Use a 'nonBlockingScope'
   * to avoid waiting if no clients are requesting a connection.
   *
   * "get???Connection" exits a reference to a "???Socket" associated
   * with the new connection. You may use this like:
   *
   *   mySocketGenerator.getStreamConnection -> aStreamSocketRef[];
   *
   * If you want to work with a specialization of the basic socket
   * patterns, extend the virtuals 'streamSocketType' and/or
   * 'binarySocketType'.
   *)
  (#
    (* basics *)
    streamSocketType:< streamSocket;
    binarySocketType:< binarySocket;
    withIdleAndPE:
      (# error:< hiErrCB (* operation level error callback *)
           (#
           do INNER;
              …
           #);
         loErrCB: errCB (*superpattern for concrete error callbacks*)
           (#
           do INNER;
              …
           #);
         usageError:< loErrCB(# do INNER #);
         resourceError:< loErrCB(# do INNER #);
         accessError:< loErrCB(# do INNER #);
         addressError:< loErrCB(# do INNER #);
         connBrokenError:< loErrCB(# do INNER #);
         intrError:< loErrCB(# do INNER #);
         internalError:< loErrCB(# do INNER #);
         unknownError:< loErrCB(# do INNER #);
         timedOut:< loErrCB(# do INNER #);
         Idle:< (# do INNER; this(socketGenerator).Idle #);
         Blocking:<(# continue: (# do true->doContinue #);
              doContinue: @boolean;
           do INNER;
              (if doContinue//false then leaveNBScope if);
              Idle;
           #);
      do INNER
      #);
    Idle:< Object; (* every local 'Idle' executes this global one *)

    (* operations *)
    getPortableAddress:
      (# addr: ^portablePortAddress;
         dopart: @...;
      do dopart
      exit addr[]
      #);
    bind: withIdleAndPE
      (# dopart: @...;
      enter port
```

```
        do dopart
        #);
      close: withIdleAndPE
        (# dopart: @...;
        do dopart
        #);
      getStreamConnection: withIdleAndPE
        (# sock: ^streamSocketType;
           timeout: @integer;
           dopart: @...;
        enter timeout
        do dopart
        exit sock[]
        #);
      getBinaryConnection: withIdleAndPE
        (# sock: ^binarySocketType;
           timeout: @integer;
           dopart: @...;
        enter timeout
        do dopart
        exit sock[]
        #);
      forceTimeout: @
        (# dopart: @...
        do dopart
        #);
      usageTimestamp: @integerValue
        (#
        ...
        #);

      (* nonBlockingScope support *)
      (* don`t 'leave' a 'nonBlockingScope'. Use 'leaveNBScope'. *)
      nonBlockingScope: (# do ... #);
      leaveNBScope: (# do ... #);

      (* socket level error callback *)
      error:< hiErrCB(# do INNER #);

      (* attributes *)
      host: @assignGuard(# t: @text; enter t exit t #);
      port: @assignGuard(# rep: @integer enter rep exit rep #);
      inetAddr: @assignGuard(# rep: @integer enter rep exit rep #);

      (* private *)
      private: @...;
   #)
```

References

[Knudsen 94] J. L. Knudsen, M. Löfgren, O. L. Madsen, B. Magnusson (eds.): *Object-Oriented Environments – The Mjølner Approach*, Prentice Hall, 1994, ISBN 0-13-009291-6.

[Madsen 93] O. L. Madsen, B. Møller-Pedersen, K. Nygaard: *Object-Oriented Programming in the BETA Programming Language*, Addison-Wesley, 1993, ISBN 0-201-62430-3

[MIA 90-1] Mjølner Informatics: *The Mjølner BETA System: – Overview*, Mjølner Informatics Report MIA 90-1.

[MIA 90-2] Mjølner Informatics: *The Mjølner BETA System: BETA Compiler Reference Manual* Mjølner Informatics Report MIA 90-2.

[MIA 91-20] Mjølner Informatics: *The Mjølner BETA System – Persistent Store*, MjølnerInformatics Report MIA 91-20.

[MIA 94-24] Mjølner Informatics: *The Mjølner BETA System – The Mjølner BETA System Tutorial,* MjølnerInformatics Report MIA 94-24.

[MIA 93-25] Mjølner Informatics: *The Mjølner BETA System – Distribution* MjølnerInformatics Report MIA 94-25.

[MIA 94-26] Mjølner Informatics: *The Mjølner BETA System – BETA Language Introduction* MjølnerInformatics Report MIA 94-26.

Index

The entries in the index with *italic* pagenumbers are the identifiers defined in the public interface of the libraries:

The minor level entries refer to identifiers defined local to the identifier of the major level entry. For those index entries referring to patterns with super- or subpatterns within the library, these patterns are specified in special sections of the minor level index for that identifier.

Entries with plain pagenumbers refer to the text of this manual.