The Mjølner BETA System Distribution

Reference Manual

Mjølner Informatics Report
MIA 93-25(1.2)
August 1996

Contents

1. Introduction

This reference manual describes the BETA distribution library, version 1.2. The distribution library allows easy implementation of distributed BETA programs exchanging object references across process and machine boundaries.

This manual describes the usage of the library through descriptions of a number of demo programs. In addition the manual is a reference manual, and provides the necessary technical details to be able to use the library efficiently. For a description of the general distribution model on which the library is based, see [Brandt 93]. For implementation details, see [Brandt 94].

2. Model Overview

This section provides an overview of the distribution model used in the BETA distribution library.

The distribution library provides the `shellEnv` application framework for developing distributed BETA programs. Methods in remote objects can be called the same way as methods in local objects. The programmer need not worry about low level communication aspects since `shellEnv` handles these automatically.

Client/Server

As a special case, the BETA distribution library makes it easy to implement client/server applications. Clients and servers are simply BETA objects residing in different processes, possibly on different network hosts. However, the client/server model is asymmetric in the sense that servers are not allowed to call methods in their clients. In contrast, with the BETA distribution library object relationships are not constrained to be asymmetric. This means that the server may call methods in the client as well as the client may call methods in the server.

The following sections provide a non-technical overview of the basic building blocks in the BETA distribution library. Later sections contain a detailed description of the library, and how to use it.

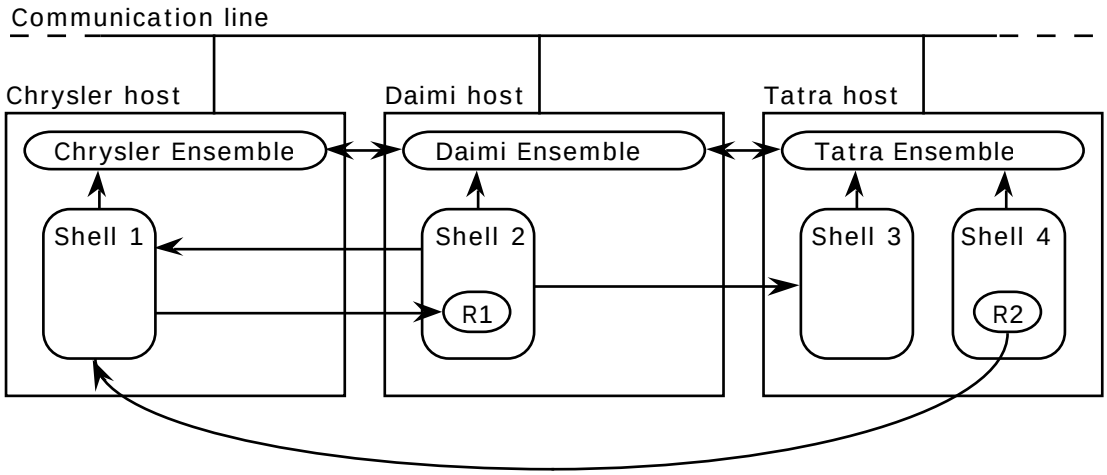

Figure 1: Ensembles, Shells and Remoteables

2.1. The basic building blocks

The main patterns of the distribution library are:

- `Remoteable`: The abstract super pattern of patterns whose instances are remotely accessible.
- `Ensemble`: An instance of the ensemble pattern represents the operating system of some network host. That is, logically there is exactly one ensemble instance for each network host[1].
- `Shell`: Each instance of the `Shell` pattern represents a single physical address space. A `Shell` thus corresponds to an operating system process.
- `NameServer`: Nameservers provide a way for distributed BETA programs to make their services available to other BETA programs. A server can provide a service by giving an object reference to some name server, i.e. registering the service in the name server. Clients may then contact the name server to lookup services required.
- `ErrorHandler`: The `errorHandler` provides a scope in which exceptions raised by communication errors may be caught and handled gracefully.

Figure 1 shows an example of a distributed system with three ensembles. The arrows symbolize object references. For example, a shell instance knows the ensemble on which it is running by having a reference to the ensemble object.

2.2. The Remoteable pattern

entry

Objects whose methods can be called from objects residing in other processes must be instances of remoteable subpatterns. Remoteable provides a nested pattern, the `entry`, to be used as a prefix for remotely called methods. For example, a remotely accessible calculator could be defined as follows:

```
calculator: remoteable
  (# plus: entry
       (# a,b,c: @Integer;
       enter (a,b)
       do a+b -> c
       exit c
       #);
     ...
  #);
```

In order for a distributed BETA program to provide a calculator service, or some other network service, the program must create an instance of the `calculator` the usual way. Afterwards the `calculator` reference can be registered in some nameserver or sent directly to a client shell.

2.3. The NameServer pattern

All object-oriented systems supporting multiple processes need a way to handle the exchange of object references. In distributed BETA, the approach is the provision of an abstract superpattern, the `NameServer`. In short, a name server simply provides a mapping between textual names and object references. The exact semantics of this mapping depends on the actual subpattern of the abstract `NameServer` superpattern[2].

The `put` operation takes a textual name and an object reference as parameters, and saves the association for later retrieval (i.e. registering the object reference).

The `get` operation in turn retrieves an object reference given its associated name and a pattern reference used for type checking.

Finally, `remove` deletes an object registration.

---

[1] However, the distribution library allows several ensembles to coexist. See section 5.4.

[2] An abstract superpattern is a pattern supposed to be used only as a superpattern, and not instantiated itself.

Example BETA code exporting a `calculator` reference to a name server (`ns`) is shown below.

```
c: ^calculator;
ns: ^NameServer;
...
(c[],"Simple Calculator")->ns.put;
```

2.4. The Shell

The `Shell` pattern is the superpattern of all objects that may be instantiated directly by an ensemble, and may in most respects be thought of as modeling an executable. An instance of the `Shell` pattern corresponds to an operating system process with an address space of its own. Non-shell objects reside in the address space provided by some shell. The `Shell` is also a remoteable[3], and may therefore provide services to remote objects. As seen from other, possibly remote, objects, a Shell instance is just another kind of object with a well defined interface.

myEnsemble

A `Shell` instance knows the ensemble on which it is running through the `myEnsemble` reference.

2.5. The Ensemble

Since the `Ensemble` is an abstraction modeling the operating system of a network host, there is a one-to-one correspondence between ensemble instances and network hosts. Below an abstract presentation of the ensemble pattern is shown.

```
ensemble: shell
  (# ...
     createShell:
       (# ...
          executableName: ^Text;
          type: ##Shell;
          sh: ^Shell;
       enter (executableName[],type##)
       do ...
       exit sh[]
       #);
     ns: @NameServer(# ... #)
  #)
```

Figure 2: Abstract presentation of ensemble

Ensemble NameServer

Part of the ensemble is a name server used as the starting point for distributed BETA processes to get in contact other processes. In a distributed system, there may be any number of network hosts, and in addition to mapping of textual object names to object references, the `NameServer` part `ns` has the responsibility of binding ensembles together. `ns` handles this responsibility by having default knowledge of other ensembles in the distributed environment. This means that if the `ns.get` operation is given e.g. the internet name of a network host, a reference to the corresponding ensemble instance is returned.

CreateShell

An important operating system task is the management of processes. Therefore, an important `Ensemble` attribute is the ability to create new `Shell` instances dynamically on the operating system represented. Ordinary BETA objects are created by instantiating patterns. However, as shells are operating system processes, they are created by instantiating an executable. The `createShell` attribute of `ensemble` takes the name of an executable and a pattern variable describing the subpattern of `Shell` of which the new shell process is expected to be an instance.

---

[3] `Shell` is a subpattern of `remoteable`.

The following BETA code illustrates how to dynamically obtain a reference to the remote ensemble named `daimi.aau.dk` followed by the creation of a shell on this ensemble.

```
            daimi: ^ensemble;
            cs: ^calcServer;
         do ...
            (ensemble##,"daimi.aau.dk") -> myEnsemble.ns.get
               -> daimi[];
            ('calcServer',calcServer##) -> daimi.createShell
               -> cs[];
```

3. Example programs

The example programs described in this section are contained in subdirectories of `~beta/demo/r4.0/distribution`.

3.1. Calculator

CalcServer
CalcClient

The `calcServer` shell is a simple server shell containing a number of calculators. Each calculator is doing work for a single client. The example includes two different clients; an `X` based client, `XcalcClient`, using the `Athena` widget set, and a non `X` client, `calcClient`. To create the executables of the calculator example, copy the contents of the calculator directory to a directory of your own and compile:

```
beta calcClient calcServer XcalcClient.
```

Before running the calculator example, make sure that an `ensembleDeamon` is running on the host(s) involved (See section 5.1).

Now execute a `calcServer` on some network host, followed by the execution of a `calcClient`. The `calcClient` prompts for the name of the ensemble where it should lookup the `calcServer`. Answer with the name of the network host on which the calcServer is running.

If a `calcServer` is not started explicitly, as was the case above, or if the `calcServer` has terminated[4], the `calcClient` will fail to lookup a `calcServer` on the host whose name it was given. In this case the `calcClient` will try to start a new `calcServer`. For this to be possible, the following changes to the calcClient code must be made:

1. Edit the `defaultAppsDir` virtual in `calcClient.bet` and `XcalcClient.bet`. If, for example, the source files were copied to the directory `mydir`, edit `calcClient.bet` as follows:

   ```
   shellEnv
   (# defaultAppsDir::
        (# do 'mydir/$/' -> dir[] #);
      ...
   #)
   ```

2. Recompile `calcClient` and `XcalcClient`
3. Move the `calcServer` executable to the machine specific subdirectory of `mydir`, e.g. `mydir/sun4s` on a SPARC running Solaris 2.x.

With the above changes, the `calcClient` is able to create a new `calcServer` if one is not already running.

---

[4] The `calcServer` terminates itself when no new calculator has been created for 5 minutes.

3.2. TimeServer

timeServer
timeClient

This example consists of a `timeServer` with a number of clients. At startup a client registers itself with a `timeServer`. When this is done, the `timeServer` will call a method in the `timeClient` once each second. Note that this means that in the usual RPC sense, the `timeServer` is now a client of its `timeClients`, since it is the server that executes methods in the clients.

`timeServer.bet` has no user interface while `timeClient.bet` is a simple `X/motif` program displaying the wall-clock time. Furthermore, for the benefit of users who do not have `X/motif` libraries, `simpleTimeClient.bet` is equivalent to `timeClient.bet`, but instead writes the wall-clock time to the console.

To run this example, copy the programs to a directory of your own and compile:

```
beta timeServer timeClient simpleTimeClient
```

Now execute a `timeServer` on some network host (remember that the `ensembleDeamon` should be running already), and finally execute some number of `timeClients`. As command line parameter, the `timeClient` takes the name of the ensemble on which the `timeServer` is expected to run. Default is the local ensemble, if no parameters are given.

3.3. Xtalk

This is a BETA distribution example, similar to the standard UNIX talk program. However, `xtalk` is an `X/motif` program. Two users each running an `xtalk` instance may connect and exchange messages by writing in a simple motif text editor.

4. Reference Manual

The interface to the distribution library is contained in the following files:

1. `~beta/distribution/v1.2/basicshell.bet`
2. `~beta/distribution/v1.2/shell.bet`
3. `~beta/distribution/v1.2/xshell.bet`
4. `~beta/distribution/v1.2/guienvshell.bet`
5. `~beta/distribution/v1.2/remoteRefAsText.bet`

All distribution specific patterns are contained in the file `basicshell.bet` and the discussion below thus describes these patterns.

- `shell.bet` should be used as `ORIGIN` in programs not using X window libraries.
- `xshell.bet` should be used as `ORIGIN` in programs using the X window libraries.
- `guienvshell.bet` should be used as `ORIGIN` in programs using the `guienv` user interface library.
- `basicshell.bet` should be used as `ORIGIN` in library fragments that must be usable in all of the above cases. The `shellEnv` pattern and all its nested patterns are contained in this file.
- `remoteRefAsText.bet` contains patterns for converting remote object references into a textual format and vice versa.

4.1. The shellEnv pattern

The `shellEnv` pattern from `basicshell.bet` is a subpattern of `systemEnv` from `~beta/basiclib/v1.5/basicsystemenv.bet`. As a consequence, distributed programs using `shellEnv` must have a structure corresponding to that of concurrent

programs using systemEnv. This structure is sketched in figure 3, showing a template for a distributed BETA program not using X window libraries.

```
ORIGIN '~beta/distribution/v1.2 /shell';
--- program:descriptor ---
shellEnv
(# shellType:: (# ... #)
   ...
#)
```

Figure 3: Template shellEnv program not using X libraries.

As can be seen from figure 3, the program:descriptor SLOT must contain a specialization of shellEnv. The usage of the shellType virtual is described in the section on the shell pattern.

> **When writing distributed BETA programs, the** program:descriptor **SLOT in betaenv must contain a subpattern of** shellEnv**. This instance of** shellEnv **is futhermore the only one allowed.**

The template in figure 3 has ORIGIN in the shell fragment since is not using X window libraries. If the program were to use X libraries, it should have ORIGIN in the xshell fragment. There is no difference between shell.bet and xshell.bet concerning the abstractions offered. The only difference is the systemEnv implementation used. An alternative implementation is necessary to be able to combine systemEnv with the event driven X window libraries using a central event loop. To be able to install a number of callback functions used for scheduling purposes, the X implementation of systemEnv needs a reference to the xtenv instance used in the current program. This is handled by the setWindowEnv virtual as illustrated in figure 4.

```
ORIGIN '~beta/distribution/v1.2 /xshell';
INCLUDE '~beta/Xt/v1.9/xtenv';
--- program:descriptor ---
shellEnv
(# shellType:: (# ... #);
   setWindowEnv:: (# do myxtenv[] -> theWindowEnv[] #);
   myxtenv: xtenv@(# ... #);
#)
```

Figure 4: Template distributed BETA program using X libraries.

In figure 4, the myxtenv object could as well have been an instance of awenv or motifenv. What counts is that it is qualified by xtenv.

The singular shellEnv instance in a distributed BETA process is returned by the getShellEnv pattern declared in basicshell.bet. Library fragments that should be usable in programs using X as well as programs not using X, must have origin in basicshell.bet.

termination

The shellEnv pattern calls INNER when it has finished initialization. Initialization of shellEnv specializations may thus take place in the shellEnv INNER, or in the shellType INNER, to be described in section 4.3. Note that the distributed BETA program will not terminate until theShell.kill has been called, even if INNER shellEnv and INNER theShell terminates!

The following sections describe the different patterns and virtuals embedded in shellEnv.

4.2. The remoteable pattern

Remoteable is the superpattern of all objects whose methods may be executed remotely. The attributes of remoteable are:

entry: The entry pattern must be used as prefix for methods to be executed remotely, i.e. called from objects in other processes. In the current release, entry subpatterns must be non-virtual.

`ping`**:** Returns `true` if the object is currently accessible and `false` otherwise.

If an `entry` is called in a local object, the overhead is limited to a simple boolean check and an `INNER` call. Thus, remoteable entries can be used efficiently also in the local case.

Subpatterns of remoteable.entry are not allowed to be virtual[5].

This restriction may seem critical, but the problem is circumvented through an extra indirection. To see how, check the example programs.

4.3. The shell pattern

The `shell` pattern is a subpattern of `remoteable` describing executables whose instances are processes. Shells should only be instantiated through `ensemble.createShell` or as part of distributed BETA programs started from the commandline[6]. The shell pattern adds the following attributes to remoteable:

`myEnsemble`**:** A reference to the `ensemble` on which this shell is running.

`kill`**:** Kills the corresponding process.

`onKill`**:** Called before the process is killed as a result of a call to `kill`.

shellType theShell

The `shellType` virtual attribute of shellEnv may be further bound in an application program. `shellType` is used as the type of the static `theShell` partobject of `shellEnv`. This instance should be the only shell instance created in a distributed BETA program. Exporting a reference to `theShell` to a remote shell gives the remote shell the ability to kill the local process by calling the `kill` attribute of `theShell`. Furthermore, `theShell` is the object whose reference is returned from `ensemble.createShell` after the creation of a new `shell` process. A distributed BETA program is not obliged to further bind the `shellType` virtual unless it is going to export a reference to `theShell` offering more functionality than the standard `Shell` pattern[7].

non-preemptive multitasking

The `INNER` part of a shell has to pause once in a while for `shellEnv` to be able to handle incoming requests from distribution partners. This is due to the non-preemptive multitasking used. Even if `INNER` terminates, the process will not terminate before `kill` has been called.

4.4. The NameServer pattern

`NameServer` is an abstract super pattern describing objects that perform a mapping between logical object names and object references. Subpatterns of `NameServer` may perform this mapping differently. Currently the only subpattern of `NameServer` implemented is the `ns` attribute of `ensemble`.

The attributes of the `NameServer` pattern are:

`put`: Saves the association between a textual name and an object reference entered as parameters. The `overWrite` virtual is called if an object of that name is already registered. If `overWrite` returns true, the existing (name,object reference) pair is overwritten with the new one.

`get`: Given a textual name and a qualification, `get` returns the object reference associated with that name, assuming it qualifies to the type entered If no associated object is found, `notFound` is raised and `NONE` is returned. If an object with the right name, but wrong type, is found, `quaError` is raised. The type entered should be a super pattern to the pattern of the object returned.

---

5 See section 6.1 describing the reason for restricting `entry`'s to be non-virtual.

6 In practice, these cases boil down to the same thing.

7 Neither `xtalk` nor any of the `calcClients` further bind `shellType`.

`remove`: Removes the named object from the `NameServer`. `notFound` is raised if no object of that name is registered in the NameServer.

NameServer is a remoteable, but all the public operations described above do some work locally before doing calls to the (private) entrys of the NameServer.

4.5. The ensemble pattern

`Ensemble` is a representation of network hosts. It has the following attributes:

`hostname`: The host name of the network host represented.

`createShell`: Allows the creation of shells, i.e. processes, on the host represented by the ensemble. `CreateShell` is described in the next section.

`ns`: A `NameServer` subpattern with added functionality. ns is described in more details below.

Never create ensemble instances on your own. The only sound way to obtain an ensemble reference is from `shell.myEnsemble`, `ensemble.ns.get` or by transferring ensemble references between shells.

4.5.1. CreateShell

Target platform

Parameters to `createShell` are the expected qualification of the `Shell` created and the executable name from which to create it. Before using the `execName` name entered, it is appended to the value returned from the `appsDir` virtual, and `$` signs in the resulting string expanded to the name of the target platform in lower case.

A number of virtuals defined locally to `createShell` allow the passing of parameters and environment variables to the newly created process. These virtuals are described below.

`appsDir`: The `appsDir` virtual names the directory where executables are located. Default value is given by the `shellEnv.defaultAppsDir` virtual described in section 4.8. In order to override the default, further bind and assign a pathname to `dir`. The interpretation given to dir is described in section 4.8. Notice that the value of `dir` should be a full path.

`screenName`: New shells created using `createShell` have no associated standard input or standard output. As a consequence, such shells are not allowed to read from standard in, and if they write to standard out or standard error, default is to redirect the output into some "black hole". On UNIX platforms a standard black hole is `/dev/null`. This default may be changed by further binding the `screenName` virtual, and assign a file name including full path to `name`. Further binding `screenName` overrides the `defaultScreenName` virtual of the new shell created. For more information on redirecting standard output, see section 4.9 describing the `defaultScreenName` virtual.

`environment`: To set `environment` variables for the newly created shell, further bind the environment virtual and call `addEnvVar` for each environment variable to be added. For example:

```
(timeClient##,'timeClient')->myEnsemble.createShell
(# environment::
    (# do ('DISPLAY','blanche:0') -> addEnvVar #)
#) -> newClient[];
```

`parameters` should be further bound and `addParam` called for each command line parameter to be given to the new shell. For example:

```
(timeClient##,'timeClient')->myEnsemble.createShell
(# parameters::
    (# do '-serverName' -> addParam;
          'daimi.aau.dk' -> addParam;
    #)
#) -> newClient[];
```

4.5.2. CreateShell Implementation

Since the distribution library is currently available only on UNIX platforms, the description in this section is UNIX specific. The implementation of `CreateShell` on e.g. a Macintosh target is different. See also section 5 that describes system dependencies.

rsh

Below, "old shell" refers to the shell executing a `createShell`, whereas "new shell" refers to the shell being created. Assuming the new shell is created on a remote host, `CreateShell` uses UNIX `rsh` to run the `remoteStart` script on the remote host. If the `rsh` process could not be forked, the `processCreationFailed` exception is raised. Thus, `processCreationFailed` signals lack of resources on the **local** host.

remoteStart
startAsDeamon

Otherwise, if forking `rsh` succeeds, the old shell reads from the standard output of the `rsh` process, i.e. what is written by the `remoteStart` script, to determine whether things went all right. `remoteStart` determines the target machine type and then executes the machine specific `startAsDeamon` executable. If the `startAsDeamon` executable or the shell executable could not be found, the `execNotFound` exception is raised. Errors different from `processCreationFailed`, relating to process start, raise the `unknownError` exception.

Finally, if the new shell process seems to have been successfully created, the thread in the old shell thread doing `createShell` waits for a callback from the new shell. If the callback[8] is not received within the time limit set by the active `errorHandler`, a `timeOut` exception is raised in that `errorHandler`. In the mean time, other threads in the old shell are allowed to do some work.

When the callback arrives, a reference to the new shell is included as a parameter. If the new shell does not qualify to the `type` entered as parameter to `createShell`, the `typeError` exception is raised.

The above description assumed that the target host was different from the local host. If the new shell is created on the local host, `sh` is used to execute the `remoteStart` script directly. That is, `rsh` is not used in the local case, making local `createShell` calls more efficient.

4.5.3. The ensemble.ns NameServer

`get`: The primary responsibility of `ns` is to bind together all ensembles in the distributed environment. Thus, if `ns.get` is asked to look up an ensemble instance, the `name` parameter is expected to be an internet name, i.e. `mjolner.dk`, and the usual name services are used to determine the internet address of the corresponding network host. Using the expected port number of the ensemble process (see section 5.4 describing the `ensemblePort` virtual), `ns.get` is able to synthesize a reference to the ensemble requested. In addition, `ns` provides a flat namespace in which objects may be saved and retrieved using `ns.put` and `ns.get`.

`put`: Saves the association between a textual name and an object reference entered as parameters. The `overWrite` virtual is called if an object of that name is already registered. If overWrite returns true, the existing (name,object reference) pair is overwritten with the new one.

`scanNames`: Iterates over the names currently registered in the nameServer.

---

[8] The callback is of course made automatically be the system.

4.6. The errorHandler pattern

The `errorHandler` supports flexible handling of network related errors. An `errorHandler` effectively defines a dynamic scope in which network errors may be caught and handled.

dynamic chain
active errorhandler

A dynamic chain of error handlers is maintained for each `systemenv system` coroutine (see [MIA 90-08]). On entry to an `errorHandler`, default is to push the handler entered onto the front of the dynamic chain of error handlers related to the current system, thereby making it the active `errorHandler`. Likewise, when the `errorHandler` is left, it is removed from the dynamic chain of error handlers, and the previously active `errorHandler` becomes active again.

error propagation

When a network related error occurs, a corresponding exception is raised in the active `errorHandler`. If the active error handler does not further bind the exception raised, the exception is automatically propagated to the previous handler in the dynamic chain. The propagation of an exception continues until either some handler catches the exception (by further binding the corresponding `errorHandler` virtual), or until the end of the dynamic chain is reached. If that happens, the exception is propagated to the `globalHandler`. If even the `globalHandler` does not catch the error, default action is to kill the current shell. If an error occurs in a coroutine that never entered an `errorHandler`, the corresponding exception is raised directly in the `globalHandler`.

Note that by default each `system` coroutine has its own dynamic error handler chain. If the top of this chain is reached, control is passed to the `globalHandler`, and not, for example, to the handler chain of the `system` that forked the failing coroutine.

controlling propagation

As described above, default is to push a newly entered `errorHandler` onto the currently active chain of error handlers. This means that exceptions not handled by some errorHandler `eh` are automatically propagated to the `errorHandler` that was active before entry to `eh` and, if `eh` was the first `errorHandler` entered by the current `system` coroutine, errors not handled by `eh` are propagated to the global handler. However, this default may be changed by supplying an `errorHandler` (`prevHandler`) as enter parameter when entering a new `errorHandler eh`. This way, exceptions not handled by `eh` will be propagated to `prevHandler`. As an example, this may be used to transfer errors automatically from a coroutine to the creator of that coroutine.

error pattern

The `error` pattern is an abstract super pattern for all communication exception virtuals. The virtual subpatterns of `error` thus correspond to different kinds of network errors, as described in the next section.

4.6.1. Error types

The network errors handled by the subpatterns of `error` inside `errorHandler` are the following.

`connectionFailed` is raised when a message send fails.

`connectionBroken` is raised when message send succeded, but the connection to the remote shell was broken before an answer could be received.

`timeOut` is raised if the remote shell failed to answer within the time limit specified by `timeOutValue` in the active `errorHandler`. Default `timeOutValue` is to wait for ever, specified by a `timeOutValue` of -1. Further bind `timeOutValue` to limit the allowed waiting time. `timeOut` is also raised if a `createShell` request did not finish within the time limit specified by `timeOutValue`.

`serverOverload` is raised if the remote shell was busy and therefore refused to handle the request. The number of concurrently allowed requests is set by `concurrentRequestLimit` in `globalHandler` of the remote shell.

`unknownObject` is raised if the remote shell did not know the object requested. This is a consequence of the remote shell doing a `withDraw` on the object requested. Thus `unknownObject` corresponds to the detection of a distributed dangling reference.

`unknownPattern` is raised if one of the patterns needed to unpack objects was not present in the local or in the remote shell. If `unknownPattern.local` is `FALSE`, some pattern needed to unpack the request was missing in the remote shell. This means that the request has not been executed. If `unknownPattern.local` is `TRUE`, some pattern needed to unpack the answer was missing in the local shell, and thus the request has actually been executed in the server shell.

`wrongAnswer` is raised if the answer from the remote shell did not have the expected format. This could mean that the remote shell is not the one we think it is, i.e. it could be another process at the same port.

4.6.2. Error handling

abort
continue
ignore

To handle network errors, further bind the corresponding `error` virtual as listed above. Within further bindings, one of the patterns `ignore`, `continue` or `abort` should be called as the last action. Note that if there are imperatives following the call to e.g. `continue`, they will not be executed! `ignore`, `continue` and `abort` are described below.

`abort`: is default if the further binding of some `error` does not call either `ignore`, `continue` or `abort`. If `abort` is called and not further specified, the remote call that failed is aborted, and the shell killed. However, to prevent the calling shell from being killed, `abort` can be further specified. In the further specification, it is allowed to do a `leave` in the dopart. For example:

```
do myLabel: errorHandler
     (# connectionFailed::
          (#
          do abort (# do leave myLabel #)
          #);
     do server.op1; ...; server.opn;
     #);
```

To ensure proper clean up, it is not allowed to `leave` **the dopart of an** `error` **virtual outside the scope of an** `abort`**.**

That is, the following code is not allowed. In the worst case it may lead to segmentation faults. In the best case, a number of communication resources may not be released properly.

```
do myLabel: errorHandler
     (# connectionFailed::
          (#
          do leave myLabel
          #);
     do server.op1; ...; server.opn;
     #);
```

`ignore`: Terminates the failing remote call, but pretends as if the remote call succeded. Control flow continues after the remote call causing the error. For example:

```
do errorHandler
   (# connectionFailed:: (# do ignore #);
   do server1.op1; server2.op2;
   #);
```

If the `server.op1` remote call fails, control flow continues with the `server.op2` call. This may of course result in rather strange program behaviour, but is the responsibility of the programmer. It makes no sense to further specify the ignore pattern since it never calls INNER.

`continue`: Depending on the actual exception, retries or continues the operation that caused the error. For example. in the case of a `timeOut`, `continue` means that the communication subsystem will wait once again for the number of seconds specified in the `timeOutValue` virtual in effect. In the case of a

connectionFailed, connection is retried. As with ignore, it makes no sense the further bind continue.

In order to maintain the errorHandler chain, it is not allowed to leave the dopart of an errorHandler directly. Leaving an errorHandler must be done from inside the dopart of a leaveHandler:

```
do someLabel: errorHandler
   (#
   do ...;
      leaveHandler (# do leave someLabel #)
   #);
```

However, from inside the error.abort pattern, the errorHandler currently handling the error may be left directly, as was shown in the abort example above.

NOTICE: It is not allowed to `leave` **the dopart of an** `errorHandler` **outside the scope of a** `leaveHandler`**.**

If multiple errorHandlers are left simultaneously, the leaveHandler nested inside the outermost errorHandler in the dynamic call chain must be used. For example:

```
do e1: errorHandler
   (# leaveFirst: leaveHandler (# do INNER #);
   do e2: errorHandler
      (# leaveSecond: leaveHandler (# do INNER #)
      do e3: errorHandler
         (#
         do leaveFirst (# do leave e1 #);
         #)
      #)
   #);
```

4.7. The globalErrorHandler virtual

The globalErrorHandler is a virtual errorHandler specifying the default handling of network errors. Further bind globalErrorHandler to specify the global errorHandler for shellEnv. If communication exceptions are not caught by another errorHandler they will eventually propagate to globalHandler, which is an instance of globalErrorHandler. If the exception is not handled there, the process is terminated by calling theShell.kill.

Additional attributes of globalErrorHandler are as follows:

concurrentRequestLimit: The maximum number of simultaneous incoming requests to this shell that is allowed. Default (-1) corresponds to no limit. If handling a request would result in breaking this limit, the request will be ignored and the client side exception serverOverload raised.

workerPoolSize: Determines the size of the pool of workers to handle incoming requests. A worker is a collection of resources needed to handle a request. Because it is cheaper to reuse these resources than to allocate new ones, the workerPool keeps track of unemployed workers ready for reuse. A reasonable value for workerPoolSize is the expected mean number of requests handled simultaneously. If a request is always executed to end without suspending implicitly of explicitly, a single worker is adequate.

4.8. The defaultAppsDir virtual

When using ensemble.createShell to start processes, defaultAppsDir specifies the default directory (full path) on the remote host in which the executable is expected to be found.

The default value returned from defaultAppsDir is

```
/usr/local/lib/beta/distribution/v1.2 /aps/$/
```

but this value may be changed in two ways:

1. Setting the `BETALIB` environment variable. If `BETALIB` is found in the environment of the shell process, the value returned from `defaultAppsDir` is `$(BETALIB)/distribution/v1.2 /aps/$/`.
2. By further binding `defaultAppsDir` and setting `dir` to whatever full path is convenient:

```
shellEnv
(# ...
   defaultAppsDir::<
     (#
     do '/myhome/myapps/$/' -> dir[];
     #)
#)
```

Alternatively the directory may be changed individually on each `createShell` call by further binding the `ensemble.createShell.appsDir` virtual.

$ expansion

Similar to specifying `INCLUDE` and `BODY` paths in the BETA fragment system, you may use `$` to specify machine dependent executable paths. That is, assume `dir` is assigned the value `/mydir/$/`, and the remote host on which the new shell is to be created is of type `sun4s`. Then, the `execName` parameter to `createShell` is appended to `dir`, and all occurrences of `$` in the resulting string is replaced by `sun4s` before using the resulting text string as the full path of an executable.

4.9. The defaultScreenName virtual

stdout, stderr

Instances of `shellEnv` created by `ensemble.createShell` cannot use the standard inputs and outputs of the process. By default standard output is redirected to `/dev/null` on UNIX.

To redirect output to a file, specify a filename by further binding `defaultScreenName`. Alternatively `screenName` may be set individually for created shells by using the `ensemble.createShell.screenName` virtual. This allows the creator to override the `defaultScreenName` of the shell created.

`stdout` as well as `stderr` are redirected to the file named in `defaultScreenName` or in `ensemble.createShell.screenName`. Since output on `stdout` and `stderr` from remotely started shells should normally be restricted to debugging output, `stdout` and `stderr` are unbuffered in order to ensure that all output is actually written to the file specified, and in the order the output was written to `stdout` respectively `stderr`. If a shell is started from the commandline, `defaultScreenname` has no effect, since `stdout` and `stderr` are then used without modification.

stdin

Shells started by `createShell` have no `stdin`, and is thus not allowed to read from standard input.

4.10. The withdraw pattern

distributed garbage collection

Due to the lack of distributed garbage collection, we need a way to explicitly withdraw the possibility of remote access to objects whose reference has crossed the shell boundary. That is, whenever a reference to a local object crosses the shell boundary for the first time, the object reference is saved in an internal table and is therefore never garbage collected. Calling `withdraw` with a local remoteable instance whose reference has been exported deletes the object from the internal table, thereby making it possible to garbage collect the object unless other local references exist. If a request to a withdrawn object arrives from a client, the request will fail with the `unknownObject` exception raised at the client side. This corresponds to following a distributed dangling reference, and cannot be avoided without distributed garbage collection.

4.11. Tracing object serializations

debugging

When performing a remote invocation, one or more objects are serialized to be sent across the network connection. In some cases, large object graphs are serialized this way. Currently there is no way to specify a limitation on the serialization traversal (as is possible in the persistent store), and sometimes more objects than expected gets serialized, leading to unexpected errors. Most often the error message resulting is 'components not handled', that is triggered when trying to pack an active object (i.e. a component). A number of shellEnv attributes support the debugging of problems like these. These attributes are described below.

Serialization tracing is initiated by setting the TraceSer boolean to TRUE. When this has been done, the BeforeSer, AfterSer and AfterUnser virtuals are called as described below:

BeforeSer is called just before an object is about to be serialized, either as a result of being sent in a remote request, or as a result of being returned as a result parameter. The object about to be serialized is given as parameter.

AfterSer is called when the object has been serialized.

AfterUnser is called when some object received, either as part of an incoming call, or as part of a the result received, has been unserialized.

Remoteable instances are not actually serialized. Instead a network representation of the corresponding object reference is sent. In case of a non-remoteable, the object is serialized and all references it contains followed.

By further binding these virtuals, it is possible to trace what objects are serialized in remote calls. A simple dump of an object may be achieved using the printObject method found in ~beta/sysutils/v1.4/objinterface.bet.

5. System Dependencies

This section describes the system files and executables needed for distributed BETA programs to work. Apart from subsection 5.1 that describes how to start the ensembleDeamon, the rest of this section describes non-standard ways of tailoring the distribution library. Therefore, most of this section can be skipped on first read of this manual.

The $ sign is used to denote the actual machinetype. For example, on a Sun SPARC running Solaris 2.x, $ should be read as sun4s.

5.1. The ensembleDeamon

In order to run any distributed BETA program, a network host must run an instance of the ensembleDeamon executable. This deamon implements the NameServer (ensemble.ns) functionality of the ensemble as well as being responsible for generating world-wide unique object ID's for distributed BETA objects created on the host on which the deamon is running.

The ensembleDeamon is installed by running the startensemble script, residing in ~beta/bin/startensemble. The ~beta/bin directory should be in the path of all BETA users. The startensemble script simply executes the correct ensembleDeamon executable for the current hardware and operating system platform[9].

The ensembleDeamon has no network security critical responsibilities and has no need for super user privileges. Therefore it makes no difference what user starts the

[9] These executables, named "MACHINETYPE"ensembleDeamon, are located in the ~beta/distribution/v1.2 directory, and may also be executed directly.

deamon. However, like other deamons `ensembleDeamon` might as well be installed by `init` when the operating system is bootet.

Section 11 contains a complete distributed BETA program that may be used to check whether an ensemble is already running on some network host.

5.2. CreateShell dependencies

For the `Ensemble` operation `createShell` to work, the owner of the executing process must have execution rights to the script

```
~beta/distribution/v1.2/private/external/remoteStart
```

and the

```
~beta/distribution/v1.2/private/external/$/startAsDeamon
```

executable on the target host. By default these executables are installed with execution rights for everyone.

remoteStart

startAsDeammon

rsh

`remoteStart` is a simple shell script determining the current machine type before invoking the machine specific `startAsDeamon` executable, which in turn executes the shell executable as a deamon. `remoteStart` is executed using `rsh` (`remsh` on HP UX platforms) on remote network hosts, or executed directly by the distributed BETA process, in case the new shell is started on the local host. If `rsh` is not found in the standard directory for the platform, the `rshPath` virtual described in section 5.5 can be used to tell `shellEnv` about the location of the `rsh` executable.

5.3. The distributionDir virtual

When using `ensemble.createShell` to start shell processes, `createShell` needs to know the location of the `remoteStart` script and the `startAsDeamon` executable on the target host. Default location of these scripts is in a subdirectory of the directory containing BETA distribution source files.

In the current MBS installation, `remoteStart` is by default placed in the `~beta/distribution/v1.2/private/external` directory, which is also the default value returned from `distributionDir`[10].

However, when shipping applications, as described in section 5.6, it is convenient to be able to put these system executables into another directory chosen by the application developer.

As for `defaultAppsDir`, `distributionDir` is changed by either setting the `BETALIB` environment variable, or by further binding `distributionDir` and assign a full path to `dir` within the further binding.

5.4. The ensemblePort virtual

communication port

This release of the BETA distribution library uses TCP/IP for communication between distributed BETA processes. However, the only system port number "hardcoded" into the distribution library is the port number assigned to the ensemble shell, i.e. the `ensembleDeamon` process started using the `~beta/bin/startensemble`.

By default, the ensemble uses the port number 5188. However, in order to allow several ensemble instances to run on the same host without conflicting, e.g. in order to allow different groups to run BETA distribution without sharing the ensemble, this may be changed in one of the following ways:

BETA_ENSEM-BLE_PORT

- Set the `BETA_ENSEMBLE_PORT` environment variable before starting the distributed program that should use an alternative ensemble port number. For example:

[10] In future releases, all system executables will be placed in the `~beta/distribution/v?.?/bin` directory.

```
setenv BETA_ENSEMBLE_PORT 5211
```

Notice that this should also be done before starting the `ensembleDeamon` to use the alternative port number.

Remember that the `ensemble.createShell.environment` virtual may be used to set the environment of shells started using `createShell`.

- Alternatively, further bind the `ensemblePort` virtual. For example:

```
shellEnv
(# ensemblePort:: (# do 5211->value #);
...
#)
```

5.5. The rshPath virtual

Currently, `ensemble.createShell` depends on `rsh` in order to start new shells on remote hosts. To support systems with `rsh` installed in a non-standard directory, the `rshPath` virtual allows for the specification of the location of the `rsh` system command. `rshPath` should be specified to the full path of `rsh` (`remsh` on HP-UX). If `rshPath` is not further specified, the system default is used. Usually there should be no need for changing the `rshPath`.

5.6. Shipping distributed BETA programs

As described in previous sections, distributed BETA applications rely on the presence of the `ensembleDeamon` process. In addition, if using the `ensemble.createShell` method, the `remoteStart` script and the machine specific `startAsDeamon` executable must be present.

When shipping distributed BETA applications, these system scripts and executables must therefore be shipped along. To allow `remoteStart` and `startAsDeamon` to be located in a directory chosen by the application developer, `shellEnv` includes the `distributionDir` virtual, making it possible to specify an alternative path. `distributionDir` is described in section 5.3.

6. Further Implementation Issues

This section briefly describes the implementation of the distribution library. Only the implementation details necessary to understand how the library should be used, are explained.

proxy

The implementation of the BETA distribution library builds on the notion of *proxy* objects. This means that references to remote objects are really references to local objects representing the remote object, as illustrated in figure 5.

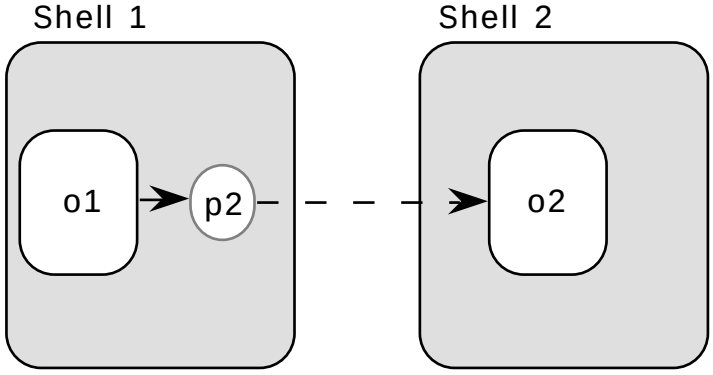

Figure 5: The proxy concept.

Proxies are automatically instantiated by the distribution library when a reference to a remote object enters a shell for the first time. Likewise, unused proxies are automatically garbage collected. In figure 5, the object `o1` has a reference to the remote object `o2`. In practice this means that `o1` has a reference to a local proxy `p2` representing `o2`. Method calls to `p2` are automatically redirected to `o2` by sending the request across the network.

6.1. Proxy object types

In principle the type of a proxy object is exactly the same as the type of the object it represents. It would be pretty impractical, however, if the current implementation demanded this, for at least the following reasons[11]:

- The proxy would have the same size as the real object had before initialization.
- As dynamic linking is not currently supported[12] the full code corresponding to the implementation of a pattern would have to be linked in with every client using the service provided by that pattern. Even with dynamic linking it would not be a feasible solution to link in server code in the client process.

The solution used in the current release of BETA distribution is to instantiate proxy objects as instances of the most specific pattern known to the client and that is declared in the `shellEnvLib:attributes` SLOT of `shellEnv`. This means that all operations to be used remotely must be declared in this SLOT, but the actual object may be an instance of any subpattern declared where it might be appropriate.

One unfortunate side effect of the implementation described above is that virtual entry's are not supported. This has to do with the implementation of virtual dispatch in BETA, and is outside the scope of this manual. However, keep in mind that

Subpatterns of `remoteable.entry` are not allowed to be virtual.

6.2. Parameter semantics

The current release is restricted in the sense that only instances of `remoteable` may be accessed remotely. This has implications for the parameter semantics of reference parameters if the referred objects are not remoteables. The parameter semantics when executing `entry's` of remote objects are described below.

Before an entry in a remoteable is executed, it is first checked whether the enclosing remoteable is a proxy or not. If it is not, the do part of the entry is executed as in any ordinary object invocation:

```
entry:
  (#
  do (if isProxy then
        <<SLOT remoteableCallremote:descriptor>>
        else INNER
     if);
  #)
```

If the remoteable is a proxy, a copy of the transitive closure[13] of the entry to be executed is created on the receiver side before the receiver side copy of the entry is executed. After execution, the transitive closure of the entry is sent back and corresponding objects updated. If the transitive closure of the `entry` object contains `remoteable` instances, these are not copied, but instead a proxy is created on the receiver side.

---

[11] Note that none of these reasons are principal problems, but they are practical problems in the current release.

[12] It will be in a future MBS release.

[13] I.e. the set of all objects reachable from the entry object.

Below a formal description of the parameter semantics concerning non-remoteables in the transitive closure of the entry is given. Note that the parameter semantics of remoteables in the transitive closure are exactly as ordinary reference parameter semantics. The semantics of other types of reference parameters are a consequence of trying to support all kinds of parameters despite the current restrictions, and are shown here to give an understanding of why, in most cases, it is possible to use any kind of parameters as usual, and why it may result in unexpected behaviour in some cases if not used with caution. The semantics are thus a compromise between on one side supporting all kinds of parameters and on the other side preserving usual BETA parameter semantics also in the distributed case.

Now, denote by *trans(e,s,t)* the set of objects reachable through object references from the entry *e* in the shell *s* at time *t*. The set *trans(e,s,t)* is built by doing a traversal of the object graph rooted in *e*, and following all references that do not reference a `remoteable` instance, the `BETAENV` object or the singular `ShellEnv` object.

Let *state(o,t)* denote the state of the object *o* at time *t*. Furthermore, define the relation (read *is the cause of*) by

$a \quad b \Leftrightarrow b$ was created as a copy of a

The parameter semantics operates with four sets of objects. Below *e* denotes the entry object about to be executed at the sender side. Note that object state as denoted by *state* only changes during execution of the entry *e´*.

A: The transitive closure of *e* immediately after assignment of the enter parameters:

$$A = trans(e, sender, before)$$

B: The receiver side copy of the set *A* before the receiver side copy of the entry *e*, *e´*, is executed.

$$B = trans(e', receiver, before) \quad e \quad e'$$

C: The transitive closure of *e´* at the receiver side after it has been executed:

$$C = trans(e', receiver, after) \quad e \quad e'$$

D: The transitive closure of *e* at the sender side after execution:

$$D = trans(e, sender, after)$$

Now, the parameter semantics are described by the following FOPL formulas:

1. The set *B* contains copies of all objects in the set *A*:

$$(\quad o \quad A. \quad o' \quad B.(o \quad o' \quad state(o, before) = state(o', before))) \quad |A| = |B|$$

2. The set *D* contains copies of all objects in *C*. Objects in *D* are either members of *A* (as is the entry *e*), or they are copies of objects in *C* that are not in *B*. (is exclusive or.)

$$o \quad C. \quad o' \quad D.state(o', after) = state(o, after) \quad ((o' \quad A \quad o' \quad o) \quad (o \quad B \quad o \quad o'))$$

$$|C| = |D|$$

The semantics described above have at least the following implications:

In/Out semantics

- Reference parameters referring to non-remoteables have approximately Ada-like in/out semantics. The object referred is copied before execution of the entry, and, assuming that it is still in the transitive closure of the entry, updated from the copy after execution.

Deep Copying

- If wanted, the parameter mechanism may be used to transfer deep copies of any kind of object between different shells. The notion of deep object copying may be useful, but is orthogonal to distribution. In future versions of BETA distribution,

the parameter semantics described above will hopefully be obsolete by removing the restriction that only remoteable instances may be accessed remotely. Instead direct support for object copying may be provided, but not as part of the distribution library. Instead future versions of distribution will support the notion of object movement between different shells.

- The serialization traversal stops when a `remoteable`, the `BETAENV` object or the `shellEnv` object is met.

Serialization limits

6.3. Non-preemptive scheduling

The scheduler implemented by `systemEnv` is non-preemptive, and as a consequence control is only transferred between coroutines when an explicit or implicit `suspend` is executed. Implicit suspend may be done when executing semaphore `P` operations, when calling submethods of `systemenv.monitor.entry`, on execution of `systemEnv.pause` and on communication with remote shells through submethods of `remoteable.entry`. When this happens, another coroutine may be attached to execute until it suspends implicitly or explicitly.

If any coroutine makes blocking system calls, this will block all coroutines, as the BETA process is seen as a single process from the operating system.

6.4. Communication layer implementation

The current implementation of the distribution library uses TCP/IP for communication between shells. On first communication between two shells, a socket connection is established, and this connection is reused for future communications.

The socket connections opened may be closed in response to one of the following conditions:

- An error occurs on the connection, leading to the conclusion that the connection is not sound anymore. In most situations this occurs when trying to communicate with a dead shell process.
- No more file descriptors are available in the current process. In that case the least recently used connection is closed down to make it possible to open a new connection to some other shell. The closed connection may be reopened if it is later needed.
- The connection is to an ensemble. Since any distributed BETA program that exports object references need to communicate with the ensemble, it is expected that the ensemble will communicate with a large number of processes during its lifetime. Thus, the ensemble automatically closes down connections that have not been used for some time.

7. Distribution and X libraries

Since `shellEnv` is based on `systemEnv`, and `systemEnv` is capable of cooperating with the X user interface libraries, this of course goes for `shellEnv` too. However, notice that e.g. `xshell.bet` must be used as origin for programs combining X and distribution (see section 4.1).

One minor catch should be noted when combining `X` and `shellEnv`. Consider the skeleton program below combining `motifenv` with `shellEnv`.

```
ORIGIN '~beta/distribution/v1.2/xshell';
INCLUDE '~beta/Xt/v1.9/motifenv';
--- program:descriptor ---
shellEnv
(# setWindowEnv::
     (# do myWindowEnv[]->theWindowEnv[] #);
   myWindowEnv: @motifEnv
     (# ...
     do <<Initialize myWindowEnv>>
     #);
   shellType:: (# ... #)
   ...
#)
```

The `setWindowEnv` virtual and `theWindowEnv` reference are declared in `basicsystemenv`. `myWindowEnv` does not have to be a `motifenv` instance as long as it is at least an `xtenv` instance. (one of `awenv`, `motifenv` or `xtenv`).

And now for the catch: The do-part of `myWindowEnv` is executed by `systemEnv` as soon as it has finished initialisation. Unfortunately this means that `shellEnv` has not yet been properly initialized. To avoid problems, the do-part of `myWindowEnv` should be empty, and initialisation should be done by an explicit `init` method called when `shellEnv` has been properly initialised:

```
ORIGIN '~beta/distribution/v1.2/xshell';
INCLUDE '~beta/Xt/v1.9/motifenv';
--- program:descriptor ---
shellEnv
(# setWindowEnv::
     (# do myWindowEnv[]->theWindowEnv[] #);
   myWindowEnv: @motifEnv
     (# ...
        init:
          (#
          do <<Initialize myWindowEnv>>
          #);
     do (* Nothing *)
     #);
   shellType:: (# ... #)
do myWindowEnv.init;...
#)
```

This problem will be solved in future releases of `systemEnv`.

8. Restrictions

The following restrictions should be kept in mind when writing distributed BETA programs based on version 1.2 of the distribution library. They have all been described in details in previous sections.

- Only remoteables can be accessed remotely.
- The public (remotely accessible) part of `remoteable` patterns must be declared in `shellEnvLib:attributes` SLOT of `shellEnv`.
- Parameters to remoteable `entry`'s must be attributes of the `entry` object and not attributes of e.g. the `remoteable` that is the static father of the entry.
- `Remoteable` entries should not be virtual. If they are, the patterns of the real object must be linked with the client and should be declared in the `shellEnvLib:attributes` SLOT of `shellEnv`.
- As there is no support for dynamic linking, objects transferred between shells (see section 6.2) must be instances of patterns known to both shells, i.e. the patterns

must be linked into both shells. If they are not, an `unknownPattern` exception will result.

9. Interface Description for the basicshell library

```
ORIGIN '~beta/basiclib/v1.5/basicsystemenv';
(*
 * $RCSfile: basicshell.bet,v $
 * $Revision: 1.5 $
 * $Date: 1996/06/21 10:49:48 $
 *
 * COPYRIGHT
 *       Copyright Mjolner Informatics, 1992-96
 *       All rights reserved.
 *
 * Use xshell or shell as origin for distributed BETA programs.
 *)
BODY 'private/shellBody';

INCLUDE '~beta/objectserver/v2.4/ObjectSerializer';
INCLUDE '~beta/sysutils/v1.5/envstring';
INCLUDE '~beta/process/v1.5/commAddress';
INCLUDE 'private/rpc_interface';

--- lib:attributes ---

(* GETSHELLENV
 * ===========
 *
 * Returns the unique shellEnv instance running. *)

getShellEnv:
  (# theShellEnv: ^shellEnv;
  do shellEnv## -> objectPool.strucGet
     (# init::<
          (#
          do ...
     #)#) -> theShellEnv[];
  exit theShellEnv[]
  #);

(* SHELLENV
 * ========
 *
 * When making distributed BETA programs,
 * the "program:descriptor" SLOT
 * in betaenv must be filled with a subpattern of shellEnv.
 * This is also
 * the only instance of shellEnv allowed. *)

shellEnv: systemenv
  (#

     (* SHELLTYPE
      * =========
      *
      * Furtherbind to specify the kind of Shell. *)

     shellType:< Shell;
     theShell: @shellType;
```

```
(* SHELLENVLIB
 * ===========
 *
 * Pattern definitions used as interfaces to remote
 * objects must be declared in the shellEnvLib SLOT.
 *
 * References to objects being instances of subpatterns
 * of these patterns may be exported for remote access
 * without being declared in this attribute slot, but at
 * least the superpattern containing the entrys to be called
 * remotely should be declared in ShellEnvLib, to make them
 * visible in the client. *)

<<SLOT shellEnvLib:attributes>>;

(* REMOTEABLE
 * ==========
 *
 * Superpattern of all objects that may be accessed remotely.
 *
 * Methods to be called remotely must be non-virtual and be
 * subpatterns of entry.
 *
 * ping returns true if the object is currently accessible and
 * false otherwise. *)

remoteable:
  (# entry:
       (#
       do (if isProxy //true then
              ...
           else INNER
          if);
       #);

     ping: booleanValue (# do ... #);

     (* private:
      * ======== *)

     ri: ^remoteInfo;
     isProxy: @Boolean;

     <<SLOT remoteablePrivateEntries:attributes>>;

  do INNER
  #);

(* SHELL
 * =====
 *
 * Pattern describing executables whose instances are processes.
 * Shells should only be instantiated through
 * ensemble.createShell
 * or started from the commandline.
 *
 * myEnsemble is a reference to the ensemble on which this shell
 * is running.
 *
 * kill kills the corresponding process. The onKill virtual is
 * called before killing the process.
 *
 * The INNER part of a shell has to pause once in a while for
 * shellEnv to be able to handle incoming requests.
 * This is due to the non-preemptive multitasking used.
```

```
 *
 * Even if INNER terminates, the process will not terminate
 * before kill has been called.
 * (Of course nasty signals may do the job.) *)

shell: remoteAble
  (#
     myEnsemble: ^ensemble;
     onKill:< Object;

     kill: (# do ... #);

     ShellPrivate: @...;

     <<SLOT shellPrivateEntries:attributes>>;

  do INNER;
  #);

(* NAMESERVER
 * ==========
 *
 * Performs mapping between logical object names and object
 * references. Subpatterns may perform this mapping differently.
 *
 * put saves an object reference under the name given.
 * The overWrite virtual is called if an object of that name is
 * already registered.
 * If overWrite returns true, the existing (name,objectref) pair
 * is overwritten with the new one.
 *
 * get looks for an object with the given name and type. If no
 * matching object is found, notFound is called.
 * If an object with
 * the right name, but wrong type is found, quaError is raised.
 *
 * remove undoes put.
 *
 * NameServer is a remoteAble, but all public operations does
 * some work locally before calling remote. *)

NameServer: remoteAble
  (#
     elementType:< RemoteAble;

     put:<
       (# overWrite:< BooleanValue;
          name: ^Text; obj: ^elementType;
       enter (obj[], name[])
       do INNER
       #);

     get:<
       (# notFound:< Notification;
          quaError:< Exception;
          name: ^Text; type: ##object;
          obj: ^elementType;
       enter (type##, name[])
       do INNER
       exit obj[]
       #);

     remove:<
       (# notFound:< Notification;
          name: ^Text;
       enter name[]
```

```
        do INNER
        #);

      <<SLOT NameServerAttributes:attributes>>;

   do INNER
   #);

(* ENSEMBLE
 * ========
 *
 * A representation of network hosts.
 *
 * hostname is the hostname of the host represented.
 * (surprise surprise)
 *
 * createShell allows creation of shells on the host represented.
 *
 *  The executable name without path is given by the "execName"
 *  parameter, and the expected type of the shell created by
 *  "instances"
 *   of this executable is given by the "shellType" parameter.
 *
 *   appsDir is the directory where executables are expected to
 *   be found.
 *   You should also check the description of the
 *   "shellEnv.defaultAppsDir" virtual.
 *
 *   screenName names the redirection file for screen output from
 *   the new
 *   shell. If screenName is not furtherbound, screen output is
 *   redirected
 *   to the file specified in the shellEnv of the shell created.
 *
 *   The execNotFound exception is raised if the executable could
 *   not
 *   be found.
 *
 *   processCreationFailed is raised if the process could not be
 *   created.
 *
 *   typeError is raised if the shell created does not have the
 *   expected type.
 *
 *   Other kinds of errors raises the unknownError exception.
 *
 *   Furtherbind environment and call addEnvVar for each
 *   environment
 *   variable to be added to the environment of the new shell.
 *
 *   Furtherbind parameters and call addParam for each command
 *   line parameter to be given to the new shell.
 *
 * ns is a NameServer with default knowledge of the ensembles in
 * the distributed environment. It is therefore possible to
 * lookup
 * other ensembles using ns. Apart from this, ns provides a flat
 * namespace in which objects may be saved and retrieved using
 * ns.put
 * and ns.get. ns.scanNames may be used to iterate over the names
 * explicitly registered in the ensemble using ns.get.
 *
 * do NOT CREATE INSTANCES OF ENSEMBLE ON YOUR OWN!!. *)

Ensemble: shell
  (# hostName: ^Text;
```

```
createShell:
  (# appsDir:<
       (# dir: ^Text
       do defaultAppsDir -> dir[];
          INNER;
       exit dir[]
       #);

     screenName:<
       (# name: ^Text;
       do INNER
       exit name[]
       #);

     environment:<
       (# addEnvVar:
            (# name, value: ^Text;
            enter (name[],value[])
            do ...
            #);
          (* private: *) env: ^Text; envCount: @Integer;
       do INNER
       exit (env[],envCount)
       #);

     parameters:<
       (# addParam:
            (# value: ^Text;
            enter value[]
            do ...
            #);
          (* private: *)
          params: ^Text; paramCount: @Integer;
       do INNER;
       exit (params[],paramCount)
       #);

     execNotFound:< Exception
       (#
       do INNER; ...
       #);

     processCreationFailed:< Exception
       (#
       do INNER; ...
          if);
       #);

     typeError:< Exception
       (#
       do INNER ...
          if);
       #);

     unknownError:< Exception
       (#
       do INNER; ...
       #);

     shellType: ##Shell;
     execName: ^Text;
     sh: ^Shell;
  enter (shellType##,execName[])
  do ...
  exit sh[]
```

```
      #);

    ns: @NameServer
      (# put:: (# do ... #);
         get:: (# do ... #);
         remove:: (# do ... #);
         scanNames:
           (# current: ^Text;
           do ...
           #);
      #);

    (* private:
     * ======== *)

    <<SLOT ensembleAttributes:attributes>>;

    ensemblePrivate: @...;
  #);

(* ERRORHANDLER
 * ============
 *
 * The "error" pattern is an abstract super pattern for all
 *  communication
 * exception virtuals. The virtual subpatterns of error thus
 * corresponds
 * to different kinds of network errors.
 *
 * When an errorHandler exception is raised that is not
 * further specified, the exception is automatically propagated
 * to the
 * previous handler in the dynamic call chain. This chain of
 * errorHandlers
 * is built by pushing an errorHandler onto the front of the
 * chain when
 * the errorHandler is entered. The propagation of an exception
 * continues
 * until either some handler catches the exception (by further
 * binding the corresponding errorHandler virtual), or until
 * the globalHandler is reached. If even the globalHandler does
 * not
 * catch the error, default action is to kill the current shell
 * process.
 * By default each coroutine has its own dynamic errorhandler
 * chain.
 * If the top of this chain is reached, control is passed to the
 * global
 * handler, and not, for example, to the handler chain of the
 * coroutine
 * that forked the active coroutine.
 * If the entered errorHandler (prevHandler) is not NONE, it will
 * be
 * used as the previous handler instead of the currently active
 * errorHandler. This may be used to transfer errors between
 * different
 * coroutines.
 *
 * To gracefully handle network errors, further bind the
 * corresponding
 * error virtual. Within further bindings, one of the nested
 * patterns
 * "ignore", "continue" or "abort" should be called as the last
 * action of
 * the errorHandler. Note that if there are imperatives following
 * the
```

```
* call to e.g. "continue", these imperatives will not be
* executed!
*
*     abort:  DEFAULT!! If abort is called and not further
*              specified,
*              the remote call that failed is aborted, and the
*              shell killed. However, to prevent the shell from
*              being killed, it is allowed to further specify
*              the abort, and  do a "leave someLabel" inside the
*              further specification.
*              For example:
*
*                 do myLabel: errorHandler
*                    (# connectionFailed::
*                          (# do abort (# do leave myLabel #)#);
*                    do server.op1;
*                       ...
*                       server.opn;
*                    #);
*
*              IT IS NOT ALLOWED TO LEAVE AN ERROR VIRTUAL
*              OUTSIDE
*              THE  SCOPE OF AN ABORT INSTANCE!!!
*
*
*      ignore: Abort the failing remote call, but pretend as if
*              the
*              remote call succeded. Control flow continues
*              after the
*              remote call causing the error.
*              For example:
*
*                 do errorHandler
*                    (# connectionFailed:: (# do ignore #);
*                    do server1.op1;
*                       ign: server2.op2;
*                       ...
*                    #);
*
*              If the "server.op1" remote call fails, control
*              flow
*              continues at the "ign:" label. This may of course
*              result
*              in rather strange program behaviour. It makes no
*              sense to
*              further specify the ignore pattern since it never
*              calls
*              INNER.
*
*      continue: Retry or continue the operation that caused the
*                error.
*                Fx. in the case of a timeout, continue means
*                that the
*                communication subsystem will wait once again
*                for the
*                number of seconds specified in the timeOutValue
*                virtual in effect.
*
* The network errors handled by the errorHandler virtuals are
* described below.
*
* connectionFailed is raised when we fail to send a message to a
* remote shell.
*
* connectionBroken is raised when message send succeded, but the
* connection to the remote shell was broken before an answer
```

```
         * could be received.
         *
         * timeOut is raised if the remote shell failed to answer within
         * the time limit specified by timeOutValue. Default timeOutValue
         * is to wait for ever for answer when doing remote calls.
         *  Furtherbind to limit the allowed waitingtime.
         *
         * serverOverload is raised if the remote shell was busy and
         * therefore
         * refused to handle the request. The number of concurrently
         * allowed
         * requests is set by globalErrorHandler.concurrentRequestLimit.
         *
         * unknownObject is raised if the remote shell did not know the
         * object
         * requested. This is a consequence of the remote shell doing a
         * withDraw on the object requested. Thus unknownObject
         * corresponds
         * to detection of a distributed dangling reference.
         *
         * unknownPattern is raised if one of the objects sent to the
         * remote
         * host was an instance of a pattern unknown there (local=FALSE),
         * or if
         * the pattern of a returned object was not known locally
         * (local=TRUE).
         * In the case of unknown pattern it makes no sense to retry the
         * request.
         *
         * wrongAnswer is raised if the answer from the remote shell does
         * not have the expected format. This could mean that the remote
         * shell is not the one we think it is, i.e. it could be another
         * process at the same port.
         *
         * NOTICE!! It is not allowed to do a "leave" from within the
         * dopart of an
         * errorHandler. If it is necessary to leave the scope of an
         * errorHandler,
         * use the "leaveHandler" pattern as follows:
         *
         *    do someLabel: errorHandler
         *       (#
         *       do ...;
         *          leaveHandler (# do leave someLabel #)
         *       #);
         *
         * If multiple errorHandlers are left this way, use the
         * leaveHandler
         * nested inside the outermost errorHandler in the dynamic call
         * chain. *)

      errorHandler:
        (#
           <<SLOT errorHandlerLib:attributes>>;

           (* ERROR
            * =====
            *
            * is the abstract super pattern of all network related
exceptions.
            *)

           error:
             (# <<SLOT errorHandlerErrorLib:attributes>>;

                abort: failureAction
```

```
        (# ... #);
      continue: failureAction
        (# ... #);
      ignore: failureAction
        (# ... #);

      theObj: ^remoteAble;
      theEntry: ^theObj.entry;
      cleanup: ^EH_cleanup;

    enter (theObj[], theEntry[], cleanup[])
    do INNER
    #);

  (* NETWORK EXCEPTIONS
   * ================== *)

  connectionFailed:< E_failed;
  connectionBroken:< E_broken;
  unknownObject:< E_unknownObj;
  unknownPattern:< E_unknownPat;
  timeOut:< E_timeOut;
  serverOverLoad:< E_overload;
  wrongAnswer:< E_answer;

  (* TIMEOUTVALUE
   * ============
   *
   * Further bind and set "sec" in order to change the default
   * "wait for ever" policy. *)

  timeOutValue:< V_timeOut;

  V_timeOut: (# sec: @Integer
             do ...
             exit sec
             #);

  E_failed: error (# do ... #);
  E_broken: error (# do ... #);
  E_timeOut: error (# do ... #);
  E_overload: error (# do ... #);
  E_answer: error (# do ... #);
  E_unknownObj: error (# do ... #);
  E_unknownPat: error
    (# local: @Boolean
    enter local
    do ...
    #);

  (* PRIVATE!!
   * ========= *)

  failureAction:
    (# callInner: @Boolean
    do INNER
    #);

  EH_cleanup:
    (# todo : @Integer;
       fa: ^failureAction;
    enter (todo ,fa[])
    do INNER
    #);

  prevHandler: ^errorHandler;
```

```
      enterHandler: @...;
      leaveHandler: (# ... #);

   enter prevHandler[]
   do enterHandler; INNER; leaveHandler;
   #);

(* GLOBALERRORHANDLER
 * ==================
 *
 * Furtherbind globalErrorHandler to specify a global
 * errorHandler.
 *
 * concurrentRequestLimit is the maximum number of simultaneous
 * requests this shell will allow. -1 means no limit. If handling
 * a request would result in breaking this limit, the request
 * will
 * be ignored and the client-side exception overLoadError will be
 * raised.
 *
 * workerPoolSize determines the size of the pool of workers to
 * handle incoming requests. A worker is a collection of
 * resources
 * needed to handle a request. Because it is cheaper to reuse
 * these
 * resources than to allocate new ones, the workerPool keeps
 * track of unemployed workers ready for reuse. A reasonable
 * value for
 * workerPoolSize is probably the expected mean number of
 * requests
 * handled simultaneously. If a request is always executed to end
 * without suspending implicitly of explicitly, a single worker
 * is adequate. *)

globalErrorHandler:< errorHandler
   (# concurrentRequestLimit:< IntegerValue
        (# do -1 -> value; INNER #);
      workerPoolSize:< IntegerValue
        (# do 5 -> value; INNER #);
   #);
globalHandler: @globalErrorHandler;

(* DEFAULTAPPSDIR
 * ==============
 *
 * When using ensemble.createShell to start processes,
 * this is the default directory on the remote host in
 * which the executable is expected to be found. The default
 * may be overridden using this virtual. Alternatively the
 * directory may be changed individually on each createShell
 * call by furtherbinding the ensemble.createShell.appsDir
 * virtual.
 *
 * As is the case when specifying INCLUDE and BODY paths in the
 * BETA fragment system, you may use '$' to specify machine
 * dependent executable paths. That is, assume "dir" is assigned
 * the
 * value '/mydir/$/', and the remote host on which the new shell
 * is to
 * be created is of type 'sun4s'. Then, the "execName" parameter
 * to
 * createShell is appended to "dir", and all occurrences of '$'
 * in the
 * resulting string then replaced by 'sun4s' before it is used as
 * the full path of an executable.
 *
```

```
 * By default, the defaultAppsDir directory is
 *
 *     '/usr/local/lib/beta/distribution/v1.2/aps/$/'
 *
 * but this may be changed either by the environment variable
 * BETALIB, having default value
 *
 *     '/usr/local/lib/beta/'
 *
 * or by furtherbinding the defaultAppsDir virtual and assigning
 * to "dir". *)

defaultAppsDir:<
  (# dir: ^Text
  do '$(BETALIB)' -> expandEnvVar
     (# defaultValue::<
          (# do '/usr/local/lib/beta/' -> envvarvalue[] #);
     #) -> dir[];
     (if dir.length -> dir.inxGet // '/' then else
         '/' -> dir.append;
     if);
     'distribution/v1.2/aps/$/' -> dir.append;
     INNER;
  exit dir[]
  #);

(* DEFAULTSCREENNAME
 * =================
 *
 * ShellEnv instances created by ensemble.createShell cannot
 * use the standard outputs of the process.
 *
 * To redirect output you may specify the name of a file by
 * furtherbinding defaultScreenName. If you miss to do so,
 * output from remotely started shells is put into /dev/null.
 * Alternatively screenName may be set individually for created
 * shells by using the ensemble.createShell.screenName virtual.
 * This allows the creator to override the defaultScreenName of
 * the shell created.
 *
 * stdout as well as stderr are redirected to the file named in
 * defaultScreenName or in ensemble.createShell.screenName.
 * Since output on stdout and stderr from remotely started shells
 * should normally be restricted to debugging output, stdout
 * and stderr are unbuffered in order to ensure that all output
 * is actually written to the file specified, and in the order
 * the output was written to stdout respectively stderr.
 *
 * If this shell is started from the commandline,
 * defaultScreenname
 * has no effect, since stdout and stderr are then used without
 * modification. *)

defaultScreenName:<
  (# name: ^Text;
  do INNER
  exit name[]
  #);

(* DISTRIBUTIONDIR
 * ===============
 *
 * When using ensemble.createShell to start processes,
 * createShell
 * needs to know the location of certain scripts. The default
 * location of these scripts is in a subdirectory of the
```

```
 * directory containing BETA distribution source files.
 *
 * By default the distribution directory is
 *
 *     '~beta/distribution/v1.2/private/external'
 *
 * where ~beta is found by inspecting the BETALIB environment
 * variable. (Default value for BETALIB is
 *  '/usr/local/lib/beta').
 *
 * On order of increasing priority, the default may be changed in
 * one of the following ways:
 *
 *   1. Further binding the distributionDir virtual and assigning
 * to
 *      "dir".
 *   2. Setting the BETA_DISTRIBUTIONDIR environment variable. *)

distributionDir:<
  (# dir: ^Text;
  do '$(BETA_DISTRIBUTIONDIR)'->expandEnvVar
     (# defaultValue::
          (#
          do (* BETA_DISTRIBUTIONDIR not set.
              * If distributionDir is not
              * further bound, use default. *)
             INNER distributionDir;
             (if dir[]=NONE then
                 '$(BETALIB)'->expandEnvVar
                 (# defaultValue::
                      (# do '/usr/local/lib/beta/'
                            -> envvarvalue[] #)
                 #)->dir[];
                 (if (dir.length->dir.inxGet)<>'/' then
                     '/'->dir.append
                 if);
                 'distribution/v1.2/private/external/'
                  -> dir.append;
             if);
             dir[]->envVarValue[];
          #)
     #)->dir[];
     (if (dir.length->dir.inxGet)<>'/' then '/'->dir.append if);
  exit dir[]
  #);

(* ENSEMBLEPORT
 * ============
 *
 * This release of the BETA distribution library uses TCP/IP for
 * all
 * communication between distributed BETA processes. The only
 * system
 * port number hardcoded into the distribution library is the
 * port number
 * assigned to the ensemble shell (the "ensembleDeamon" program
 * started
 * on the local host using the "~beta/bin/startensemble" script).
 *
 * By default, the ensemble uses the port number 5190. However,
 * in order to allow several ensemble instances to run on the
 * same host without conflicting, e.g. in order to allow
 * different groups to run BETA distribution without sharing the
 * ensemble, this may be changed in one of the following ways:
 *
 *    1. Set the BETA_ENSEMBLE_PORT environment variable before
```

```
 *       starting the distributed program supposed to use an
 *       alternative port number. For example:
 *
 *           setenv BETA_ENSEMBLE_PORT 5211
 *
 *       Note that this should be done before starting the
 *       ensemble to
 *       use the alternative portnumber. Remember that the
 *       ensemble.createShell.environment virtual may be used to
 *       set the
 *       environment of shells started from other shells.
 *
 *    2. Furtherbind the ensemblePort virtual found below.
 *       For example:
 *
 *           --- program:descriptor ---
 *           shellEnv
 *           (# ...
 *              ensemblePort::< (# do 5211 -> value #);
 *              ...
 *           #)
 *)

ensemblePort:< IntegerValue
  (# valueAsText: ^Text;
  do '$(BETA_ENSEMBLE_PORT)' -> expandEnvVar
     (# defaultValue::< (# do '5190' -> envvarvalue[] #)#);
       -> valueAsText[];
     valueAsText.reset;
     valueAsText.getInt -> value;
     INNER;
  #);

(* RSHPATH
 * =======
 *
 * Currently ensemble.createShell depends on "rsh" in order to
 * start new shells on remote hosts. In order to support systems
 * with rsh
 * installed in a non-standard directory, this virtual allows for
 * the specification of the location of the rsh system command.
 * rshpath should be specified to the full path of the "rsh"
 * ("remsh" on HP UX) system command. If rshpath is not further
 * specified, the system default is used. Usually there should be
 * no need for changing the rshpath. *)

rshPath:<
  (# path: ^Text;
  do INNER
  exit path[]
  #);

(* USERNAME
 * ========
 *
 * Returns username of process owner. *)

userName: @
  (# t: ^Text;
  do (if t[] //NONE then ... if)
  exit t[]
  #);

(* WITHDRAW
 * ========
 *
```

```
 * Due to the lack of distributed garbage collection, we need a
 * way
 * to explicitly withdraw the possibility of remote access to
 * objects
 * whose reference has crossed the shell boundary. Whenever that
 * happens,
 * the object reference is saved in an internal table and is
 * therefore
 * never garbage collected. Calling withdraw with a local object
 * whose
 * reference has been exported deletes the object from the
 * internal
 * table, thereby making it possible to garbage collect the
 * object unless
 * other local references exists. If a request to a withdrawn
 * object
 * arrives from a client, it will fail with an 'unknownObject'
 * exception.
 * This corresponds to following a distributed dangling
 * reference, and
 * there is no way to avoid this without distributed garbage
 * collection.
 *
 * Proxy objects are garbage collected automatically as is any
 * ordinary
 * object.
 *)

withDraw:
  (# ra: ^remoteAble
  enter ra[]
  do ...
  #);

(* TRACING OBJECT SERIALIZATIONS
 * ============================
 *
 * When performing a remote invocation, one or more objects are
 * serialized (marshalled) to be sent across the network
 * connection.
 * In some cases large object graphs are serialized this way.
 * Currently
 * there is no way of specifying a limitation on this
 * serialization
 * traversal (as is possible in the persistent store), and
 * sometimes
 * more objects than expected gets serialized, leading to
 * unexpected
 * errors. Most often the error message resulting is "components
 * not
 * handled", which is triggered when trying to serialize an
 * active
 * object. To debug problems like these, a number of patterns are
 * offered
 * below.
 *
 * Tracing is initiated by setting the "TraceSer" boolean
 * to TRUE. When
 * this has been done, the "BeforeSer", "AfterSer"
 * and "AfterUnser"
 * virtuals are called as described below:
 *
 * BeforeSer is called just before an object is about to be
 * serialized, either as a result of being sent in a remote
 * request, or
 * as a result of being returned as a result parameter.
```

```
 *
 * AfterSer is called when the object has been serialized.
 *
 * AfterUnser is called when some object received, either as
 * part of
 * an incoming call, or as part of a the result received,
 * has been
 * unserialized.
 *
 * Remoteable instances are not actually serialized. Instead
 * a network
 * representation of the corresponding object reference is sent.
 * In case
 * of a non-remoteable, the object is serialized and all
 * references it
 * contains followed. *)

TraceSer: @Boolean;

BeforeSer:<
  (# o: ^Object
  enter o[]
  do INNER
  #);
AfterSer:<
  (# o: ^Object
  enter o[]
  do INNER
  #);
AfterUnser:<
  (# o: ^Object
  enter o[]
  do INNER
  #);

(* EVERYTHING BELOW IS PRIVATE!
 * ============================ *)

senvPriv: @...;

(* REMOTEABLETYPE
 * ==============
 *
 * The remoteAbleType is a network representation of remoteAble
 * subpatterns. The type represented includes the part of the
 * superpattern chain having origin in shellEnv, excluding
 * remoteAble as this is the basepattern for all patterns
 * represented.
 *
 * groupNames are the names of the groups corresponding to
 * groups.
 * The path of the groupNames are not included. Instead a check
 * is made
 * at startup time, that no two groups in the executable have the
 * same
 * name, as this cannot be allowed. The reason for this is to
 * avoid the
 * usual problems with pathnames, but it means that no two
 * program files
 * can have the same name.
 *
 * groups are the indices in the local execGroupTable
 * corresponding to
 * groupNames. If a groupName does not exist in the local
 * execGroupTable,
 * group will be -1.
```

```
        *
        * protos are the indices of prototypes in the groups.
        *
        * bestKnown is the most specific superpattern of the represented
        * type
        * that is known to the local shell and that has origin in
        * shellEnv.
        *
        * remoteAbleType instances are created by typeAllocator in
        * shellBody. *)

      remoteAbleType:
        (# groupNames: [1]^Text;
           groups: [1]@Integer;
           protos: [1]@Integer;
           last: @Integer;

           bestKnown: ##remoteAble;
         #);

      (* REMOTEINFO
       * ==========
       *
       * A specialization of ObjectTableElement containing address
       * information on the corresponding object ra.
       *
       * shellOID is the OID of the shell containing ra.
       *
       * shellAdr is the network address of the shell containing ra.
       *
       * netType is the network representation of the type of ra.
       *
       * ensembleAdr is the network address of the ensemble where
       * a remoteAble exists. ensembleName is the name of the ensemble.
       *
       * ensembleAdr and ensembleName are NONE unless the remoteInfo
       * corresponds to a shell or an ensemble. *)

      remoteInfo: ObjectTableElement
        (# shellOID: @OIDtype;
           shellAdr: ^portablePortAddress;
           netType: ^remoteAbleType;
           ensembleAdrAsText: ^Text;
           ensembleName: ^Text;
        enter
        (shellOID,shellAdr[],netType[],ensembleAdrAsText[],ensembleName[])
        exit
        (shellOID,shellAdr[],netType[],ensembleAdrAsText[],ensembleName[])
        #);

      initBeforeScheduler::<
        (#
        do ...
        #);

      isEnsemble:< BooleanValue;

    do ...;
      INNER;
    #);
```

10. Interface Description for the RemoteRefAsText library

```
ORIGIN 'basicshell';
BODY 'private/remoteRefAsTextBody';

--- shellEnvLib:attributes ---

(* REFASTEXT
 * =========
 *
 * Converts a remoteable reference into a text.
 * The text may later be converted into a reference
 * using refFromText.
 *
 * If the remoteble given as parameter does not already
 * have a globally unique OID, one is assigned. *)

refAsText:
  (# r: ^remoteable; t: ^Text;
  enter r[]
  do ...
  exit t[]
  #);

(* REFFROMTEXT
 * ===========
 *
 * Converts a text earlier created by refAsText back into the
 * original object reference. Doing so involves communication
 * with the Shell containing the object in order to obtain type
 * information. *)

refFromText:
  (# t: ^Text; r: ^remoteable;
  enter t[]
  do ...
  exit r[]
  #)
```

11. pingEnsemble.bet example program

```
(* pingEnsemble.bet
 * ================
 *
 * This program may be used to check whether an ensemble
 * is already running.
 *
 * Execute as:
 *     pingEnsemble <ensembleName>
 * where <ensembleName> is the name of the network host
 * on which to check for ensemble presence. *)

ORIGIN '~beta/distribution/v1.2/shell';
--- program:descriptor ---
shellEnv
(# shellType::
     (# ensembleName: ^Text;
     do (if NoOfArguments = 2 then
           'Usage: pingEnsemble <ensembleName>'->putLine;
          kill;
        if);
        1 -> arguments -> ensembleName[];
        (ensembleName[], ensemble##)
          -> myEnsemble.ns.get
          -> ensemble[];
        (if ensemble[] = NONE then
           'Network host ' -> putText;
           ensembleName[] -> putText;
           ' not found.' -> putLine;
           kill;
        if);
        (if ensemble.ping then
          'Ok.ensembleDeamon found on ' -> putText;
          ensembleName[] -> putLine;
        else
          'No ensembleDeamon on ' -> putText;
           ensembleName[] -> putLine;
        if);
        kill;
     #);
#)
```

References

[Brandt 93] Søren Brandt, Ole Lehrmann Madsen: *Object-Oriented Distributed Programming in BETA*. In Lecture Notes In Computer Science, LNCS 791, Springer-Verlag 1994.

[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 90-8] Mjølner Informatics: *The Mjølner BETA System: Basic Libraries, Reference Manual*, Mjølner Informatics Report MIA 90-8

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

[Brandt 94] Søren Brandt: *Implementing Shared and Persistent Objects in BETA. Progress Report.* Technical Report. Computer Science Department, Aarhus University.

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.