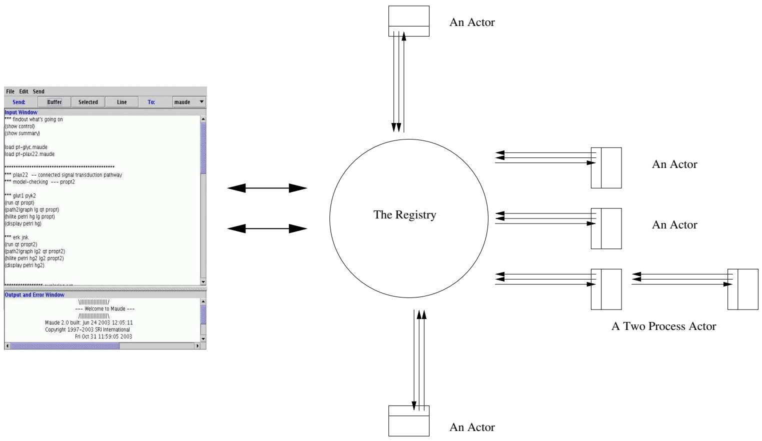
The InterOperability Platform Manual
IOP version 0.14

Ian A. Mason*
University of New England, Armidale, NSW, Australia
`iam@turing.une.edu.au`

Carolyn L. Talcott†
SRI, Menlo Park, California, USA
`clt@csl.sri.com`

January 7, 2006



*Most of the work described here was done while holding an International Fellowship at SRI, Menlo Park, partially supported by an Australian Research Council Discovery grant DP0345664, and SRI grant CCR-0234462.

†Partially supported by DARPA through Air Force Research Laboratory Contract F30602-02-C-0130, NSF grants CCR-9900326, CCR-0234462 Office of Naval Research Contract N00014-01-0837.

Contents

1 Current Version & Recent Changes

The current version of IOP is `0.14`. An earlier version of IOP (`0.08`) was described in [1] and is superseded by this one. The most recent changes in the system, in going from `0.12` to `0.14`, is the promotion of the registry to the status of an actor, called the system actor, and the stable incorporation of the Graphics 2D actor. These changes, particularly the former, have allowed the system to be configurable via instructions in the `.ioprc` file. The default now is for IOP to only start up with the minimal set of actors: the system and the GUI front end. This has the consequence that the optional Maude file on the command line is no longer supported.

The registry is the same UNIX process as the system actor, so in this sense they are synonymous. However there are more facets to the registry than just its role as the system actor, so we will not use the two terms interchangeably. Prefering the term registry to emphasize its multifaceted nature, when indeed we are talking about more than just its role as an actor in the system.

2 Command Line Arguments

The usual way to start up IOP is via the command:

```
iop
```

IOP understands three command line switches:

`-a` (default actors)

`-n` (no windows)

`-d` (debug flag on)

and of course all eight combinations. The `-a` flag will cause IOP to create at startup, in addition to the system and GUI actors, the other five actors described in section 5. It also invalidates any system actor requests that may appear in the users `.ioprc` file. The `-n` will prevent the system from starting the GUI front end, and use, instead, a minimalistic command line loop. Finally the `-d` turns on debugging flags in both the iop and registry processes, and is used for debugging purposes.

3 The General Architecture

IOP's design is based on the actor model of distributed computation [2]. IOP consists of a pool of actors that interact with one another via asynchronous message passing. The pool of actors is dynamic, it may grow or shrink as time goes by. Actors can be initial actors, created at startup, or be created by another actor already in the system in response to some event, such as an actor receiving a message, or reacting to some external action, such as a connection being made to a socket. New actors can also be created by explicitly asking the system actor to do so, by sending it a start request. Though strictly speaking this is just a special case of an actor being created in response to an event. The collection of actors created at startup is easily configurable and new actors can be designed and added to the system.

An actor in IOP usually is simply a UNIX style process that has been registered with the system according to a simple procedure. Part of this registration process involves allocating three FIFOs, or UNIX style named pipes, and redirecting the actor's `stdin`, `stdout` and `stderr` file descriptors to these special files [3].

Invoking IOP from the command line results in the following startup procedure taking place. The first process, being the `main` of IOP, parses the command line arguments, and creates the registry or system actor, the GUI actor (if the `-n` flag is absent), and the other five actors if the `-a` flag is present. After startup the `main` acts mainly as a signal handler, ensuring clean and graceful shutdown. If the `-a` flag is absent, then it is the registry that creates and configures the system according to the instructions in the `.ioprc` file. The registry keeps track of the current actors, and maintains the lines of communication between these actors. The GUI front end, pictured in figure 4, provides the user with an easy means of sending messages to any of the actors in the system. The upper part can be used to compose messages to be sent to any of the IOP actors. A file of precomposed messages can be loaded, see section 6, and message edits can be saved. The lower part displays any output from the actors that is not inter-actor communication (errors or messages to the user).

The registry maintains a list of all the actors that are registered with it. It performs several functions, and maintains three lines or forms of communication. The three forms of communication are: *inter-actor* communication, messages sent from one actor to another; *meta-actor* communication, actors notifying the registry of the birth or death of actors; and *interface* communication, communication between the registry and actors with the GUI front end. Each type of communication has a dedicated infra-structure that supports it. In the case of *inter-actor* communication, each registered actor in the system has three FIFOs, in `/tmp/`, associated with it. For each actor in the system there are three dedicated registry threads, one to monitor each FIFO that is associated with the actor's `stdin`, `stdout` and `stderr` file descriptors. The registry also has two FIFOs (again in `/tmp/`) that are used in various meta-communications, such as the registering of a newly created actor, or from an actor politely informing the system of its imminent demise. All files in `/tmp/` incorporate into their name the unique process identifier of the `main` process associated with them, hence multiple IOP's on the same machine do not interfere with one another. Finally the registry communicates with the GUI front end by using two socket connections established at startup.

Inter-actor communication is purely ASCII text, and is implemented in two layers, the *user layer*, and the *transport layer*. In the transport layer a message consists simply of a line of text representing a number (i.e an integer in base ten), followed by that specified number of bytes. The user layer, implemented on top of the transport layer, consists of the address of the target actor, the address of the sending actor, followed by the body of the message, each on a new line:

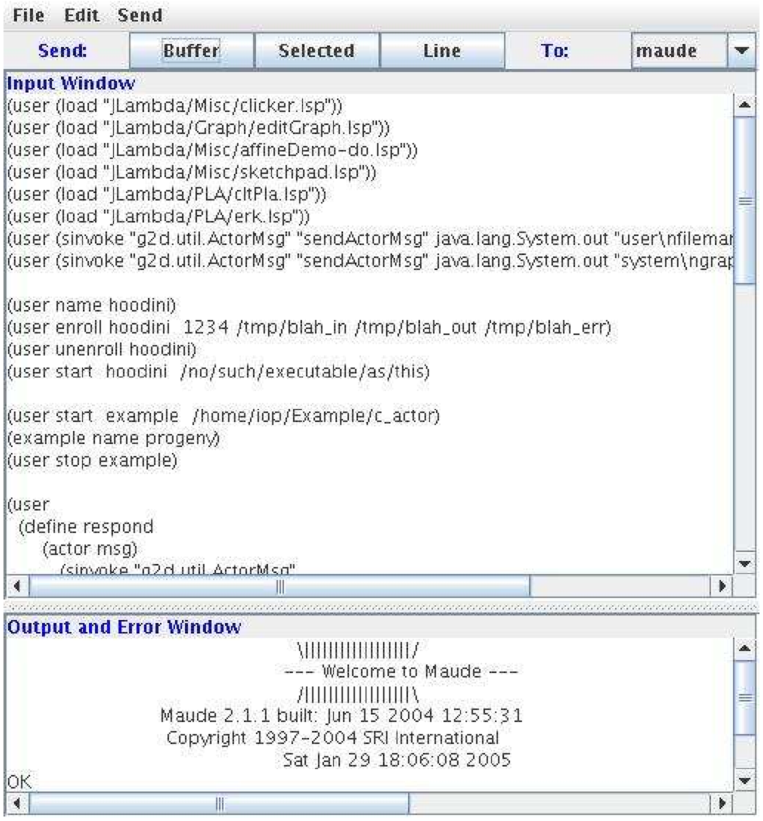
```
maude
graphics
show mauderule 25
```

This same message can be sent from the GUI by selecting `Maude` as the destination, and sending the text `(graphics show mauderule 23)`. Either way the message is transmitted in the transport layer as the sequence of bytes:

```
33\nmaude\ngraphics\nshow mauderule 25\n
```

Simple libraries implement the user layer on top of the transport layer, and allow for reliable cross platform and architecture independent communication.

4 The GUI Front End



The GUI front end, depicted in figure 4, allows the user to interact with any actor in the system. It consists, from top to bottom, of a menu bar, a button panel, the input window, and the output window. There are multiple redundancies in the design of this GUI interface. Anything that can be done with the menu bar, can also be done without it. Either by control sequences, or in the case of sending messages, by using the button panel. The menu bar can be consulted to establish, on a particular operating system, the corresponding control sequences.

The input window is a rudimentary text area allowing the user to format, and send messages to any particular actor in the system. The text sent to the chosen actor can either be a single line of text, the selected or highlighted text, of the whole buffer. Selecting the target actor is done by using the choice widget in the right side of the button panel. This can also be done programmatically in the `.ioprc` file (see section 4 for more details), or by messaging the system actor (see section 5.1 for more details).

The text in this text area can be loaded in one of three ways: manually using either the menu bar, or the control sequence associated with file loading; by specifying the full path of the file as the first line in the user's `.ioprc` file, see section 6; or automatically at startup, by naming the file `input.txt`, and placing it in the directory where you executed the `iop` command. This last method is usually the most practical. One has a directory with various files one is using for the current project, and amongst these is the `input.txt` file, that serves a role similar to a rudimentary `makefile`.

The output and error window is a non-editable text area that displays the error streams of all the actors in the system, as well as any actor message that is sent to an unrecognized actor, a name of an actor not recognized by the system. Typically any message addressed to the `user` actor will show up here, as long as the system is configured so that there is no *bona fide* actor by that name.

5 The Actors

The IOP system currently comes with seven built-in actors. They are: the system actor, the GUI actor, the maude actor, the graphics 2D actor, the executor actor, the filemanager actor, and the socketfactory actor.

These seven may all be launched with the system at start up by the command `iop -a`. Only the first two are launched by default, using the command `iop`. Only the first is *compulsory* and it alone is launched using the command `iop -n`. Alternately, each individual actor (other than the system actor) may be explicitly started up by requesting the system actor to do so. We describe actor each in turn.

5.1 The System Actor

The first major difference between version `0.12` and `0.14` of IOP is the elevation of the registry to the status of an actor in the system. This was done to enable the starting pool of actors to be easily customizable, either by directly sending the system actor, as the registry is now known, a request to either *start* or *stop* an actor. Or by describing the desired actors at startup in the `.ioprc` file, see section 6 for more details. The system actor also responds to a *select* request, which results in the specified actor being chosen as the currently selected actor in the GUI front end. Again, such a request can also be made from the `.ioprc` file. These three commands make up the *configuration interface* to the system actor. There is also a new *registration interface* that consists of three other requests. These three requests that the system actor responds to are: a *name* request, an *enroll* request, and an *unenroll* request. These requests are designed to make it relatively easy for users to program their own actors into the system. In particular, for these new actors to be able to spawn new actors in the system, by using this registration interface. We will discuss the *configuration interface* here, and leave the *registration interface* to section 7.

5.1.1 The System `start` Request

A start request to the system actor can take one of three forms. If sent from another actor it takes the form:

```
system
<sender>
start
<name> <executable> <argv[1]> ... <argv[N]>
```

If it is sent from the IOP GUI front end it takes the form:

```
(<sender> start <name> <executable> <argv[1]> ... <argv[N]>)
```

If it is requested within the `.ioprc` file, then it takes the form:

```
start <name> <executable> <argv[1]> ... <argv[N]>
```

In response to such a request, the system first finds a unique new actor name based on `<name>`[1], it then creates, and registers with the system, an actor whose executable is named by `<executable>`, whose argument array is `argv`, `argv[0]` is set to be the actor's unique name, call it `nameN`. If the system actor successfully creates a new actor, it replies with

```
<sender>
system
startOK nameN
```

If is unsuccessful it replies with:

```
<sender>
system
startFAILED nameN
```

The start request is rather robust and succeeds even if the newly created actor is stillborn, for example if `<executable>` fails to name an executable file in the file system. Down in the very depths of the implementation the new actor is created via a new process, suitably configured, executing:

[1] If `<name>` is unique as is, then this is the name chosen. Otherwise the addition of the smallest numeric suffix that makes the name unique is chosen.

```
execvp(executable, argv);
// report error and unregister here
exit(EXIT_FAILURE);
```

where `argv` is as described above.[2] If this call to `execvp` fails, an error report is sent to the GUI, and the stillborn actor is removed from the registry data structures. However, the parent actor will still respond with a success message. It is the new actor's process that sees the failure of the `execvp`, whereas it is the parent that replies to the request. Now the child *could* reply with failure, but the parent would still reply with success, since the parent doesn't see the failure. Thus `startFAILED` will only happen for pretty fatal reasons like running out of memory etc.

Some simple examples of start requests are:

```
(user start maude iop_maude_wrapper /usr/local/maude-linux/bin)
(user start maude iop_maude_wrapper /usr/local/maude-linux/bin)
(user start graphics2d iop_graphics2d_wrapper /usr/iop)
(user start filemanager iop_filemanager)
```

In the case that the newly created actor itself wishes to create actors, it will need to be able to register them with the registry. To do this it must be prepared to receive the names of the FIFOs to use. For this we use the strings `*FIFO_IN*` and `*FIFO_OUT*` to indicate where in the `argv` array, the newly created actor expects them. So for example to start the socketfactory actor, by hand, requires the following incantations:

```
(user start socketfactory iop_socketfactory *FIFO_IN* *FIFO_OUT*)
```

These wild cards are only needed in the case of actors, which themselves procreate, that are written in the earlier versions of IOP, that do not make use of the registration interface to the system actor, but rather use the meta-actor communication infrastructure. We will describe in section 7 how to write actors and incorporate them into the system.

5.1.2 The System `stop` Request

A stop request to the system actor takes the form

[2]There is, by force, one exception to this rule. If the executable is `java`, then `argv[0]` is also `java`. This is because `java` refuses to go by any other name. For this reason, if an actor written in `java` needs to know its name, then we implement it as a two process actor, the first process is a simple C wrapper process that acts as a go between. For example, this is true of the Graphics 2D actor.

```
system
<sender>
stop
<name>
```

or from the IOP GUI front end:

```
(<sender> stop <name>)
```

In response to such a request the system terminates the actor whose name is `<name>`, using the kill signal, and deregisters it from the system. Deregistering an actor involves removing it from all of the system's internal data structures, in particular any FIFOs associated with the actor are removed from the file system. If successful, it replies with

```
<sender>
system
stopOK <name>
```

If is unsuccessful, it replies with:

```
<sender>
system
stopFAILED <name>
```

Though the only reason it can be unsuccessful is if `<name>` is not recognized as a valid actor name in the system. Requesting that the system actor stop itself is the same as shutting down IOP gracefully.

5.1.3 The System `select` Request

A select request to the system actor takes on of three possible forms. As a message sent to the system actor it takes the form:

```
system
<sender>
select
<name>
```

As a message sent from the IOP GUI front end it takes the form:

```
(<sender> select <name>)
```

Or as a configuration request in the `.ioprc` file, it takes the form:

```
select <name>
```

In response to such a request the system requests that the GUI front end sets the named actor to be the selected actor. The selected actor in this sense is the actor whose name appears in the choice widget, and who is the target of any requested message sent from the GUI. There is no reply to a select request.

5.2 The GUI Actor

Currently the GUI actor accepts no actor requests. Its role in the system is purely as a graphical user interface.

5.3 The Graphics 2D Actor

The Graphics 2D Actor is simply an entry point to the interpreter of the `JLambda` language [4]. Thus the generic request takes the form

```
graphics2d
<sender>
<jlambda expression>
```

or from the IOP GUI front end:

```
(<sender> <jlambda expression>)
```

which simply results in the Graphics 2D actor evaluating the supplied expression in a separate thread of execution.[3] There is no built in response to such a request. If a request is desired, then it should be coded into the form of the expression to be evaluated. For example if one sends the following two messages to the Graphics 2D actor from the GUI front end

```
(user
  (define respond
      (actor msg)
         (sinvoke "g2d.util.ActorMsg"
                        "sendActorMsg"
                        java.lang.System.out
                         (concat actor
                                 "\ngraphics2d\n"
                                 msg
                                 "\n"))))

(user (apply respond "user" "hey!"))
```

the first will result in no response, while the second will subsequently respond with

```
user
graphics2d
hey!
```

[3]In the examples we assume that we are talking to the first actor enrolled with that name, if there were several such actors with the same name prefix, then messages would be addressed to, for example, `graphics2d<n>`

and will be displayed in the GUI's output and error window.

While the Graphics 2D actor was originally designed to process and display graphical information, its functionality far exceeds this. Since the `JLambda` language provides an interpreted interface to the entire Java class libraries, most things, if they can be done in Java, can be done by suitable requests to the Graphics 2D actor. We plan to produce `JLambda` libraries that make the remaining actors in this section largely redundant. Though there is nothing to stop the user from doing this themselves.

5.4 The Filemanager Actor

The Filemanager actor provides rudimentary access to the underlying file system. It can be asked to read from, write to, and append to files.

5.4.1 The Filemanager `read` Request

A read request to the Filemanager actor takes the following form:

```
filemanager
<sender>
read
<file>
```

or from the IOP GUI front end:

```
(<sender> read <file>)
```

In response to such a request, the filemanager attempts to open the specified file, lock it, and read its contents. If successful it replies to the `<sender>` with the appropriate contents. If it fails it logs the reason out to the error logging file and replies with a failure message.

```
<sender>
filemanager
contents <file>
<text>
```

or

```
<sender>
filemanager
readFailure
<file>
```

5.4.2 The Filemanager `write` Request

A write request to the Filemanager actor takes the following form:

```
filemanager
<sender>
write
<file>
<text>
```

or from the IOP GUI front end:

```
(<sender> write <file> <text>)
```

In response to such a request, the filemanager attempts to open the file, lock it, and write the supplied text out to the file. The file is created if it doesn't already exist. The previous contents of the file are lost. If it fails it logs the reason out to the error logging file, and replies with a failure message.

```
<sender>
filemanager
writeOK
<file>
```

or

```
<sender>
filemanager
writeFailure
<file>
```

5.4.3 The Filemanager `append` Request

An append request to the Filemanager takes the following form:

```
filemanager
<sender>
append
<file>
<text>
```

or from the IOP GUI front end:

```
(<sender> append <file> <text>)
```

In response to such a request, the filemanager attempts to open the file, lock it, and append the supplied text out to the file. The file is created if it doesn't already exist. If it fails it logs the reason out to the error logging file, and replies with a failure message.

```
<sender>
filemanager
appendOK
<file>
```

or

```
<sender>
filemanager
appendFailure
<file>
```

5.4.4 Filemanager Notes

The filename in any of the filemanager requests may be of the form `~/path`. Here ˜ will be interpreted as the home directory of the user running this instance of `iop`. All file locking is done via `fcntl`, except on Mac OS X, where it is done via `flock` because of Mac OS X `fcntl` idiosyncrasies.

5.5 The Socketfactory Actor

The Socketfactory Actor knows the number of:

- clients it has successfully created, `<clientNo>`.
- listeners it has successfully created, `<listenerNo>`.

5.5.1 The Socketfactory `openclient` Request

An open client request to the Socketfactory actor takes the following form:

```
socketfactory
<sender>
openclient
<host>
<port>
```

or from the IOP GUI front end:

```
(<sender> openclient <host> <port>)
```

In response to such a request, it attempts to connect to the specified `<host>` and `<port>`. If successful it creates a new socket actor corresponding to that socket connection. It then replies the the request with the name of the new socket actor. The created actor's name is of the form:

```
"clientsocket<clientNo>"
```

If it is successful it replies with a message of the form:

```
<sender>
socketfactory
openClientOK
<client socket name>
```

Otherwise it replies with:

```
<sender>
socketfactory
openClientFailure
```

5.5.2 The Socketfactory `openlistener` Request

The open listener Socketfactory request takes the following form:

```
socketfactory
<sender>
openlistener
<port>
```

or from the IOP GUI front end:

```
(<sender> openlistener <port>)
```

In response to such a request, it attempts to create a listening socket on the given port. If successful it creates a new listener actor (with client actor `<sender>`) that encapsulates that listening socket. The name of the listener actor is of the form:

```
"listener<listenerNo>"
```

If successful it responds with the message:

```
<sender>
socketfactory
openListenerOK
<listener name>
```

Otherwise it responds with a failure notification:

```
<sender>
socketfactory
openListenerFailure
```

5.5.3 SocketFactory Notes

The SocketFactory actor has a signal handler that waits on any child in response to a `SIGCHLD` signal delivery. This prevents the exiting of any spawned actors from remaining in the system as zombies.

5.6 The Socket Actor

The Socket Actor knows the socket that it corresponds to, and whether or not it is still open. It also keeps track of the number of requests, though this is not used.

5.6.1 The Socket `read` Request

The Socket read request takes the following form:

```
socket
<sender>
read
<no of bytes>
```

or from the IOP GUI front end:

```
(<sender> read <no of bytes>)
```

In response to such a request, if the socket is still open it attempts to read the specified number of bytes from the socket. *This is taken to be an upper limit.* If this read is successful (i.e. reads a non-zero number of bytes) it then replies with the number of bytes read, and the actual bytes read. If it fails either because the socket has been closed, or the read failed, then it logs the reason, and replies with a failure message.

If successful it replies with:

```
<sender>
socket
readOK
<no of bytes read>
<bytes>
```

or

```
<sender>
socket
readFailure
```

otherwise.

5.6.2 The Socket `write` Request

The Socket write request takes the following form:

```
socket
<sender>
write
<no of bytes>
<bytes>
```

or from the IOP GUI front end:

```
(<sender> write <no of bytes> <bytes>)
```

In response to such a request, if the socket is still open, and `<no of bytes>` is nonzero, it attempts to write the specified number of bytes to the socket. If this write is successful (i.e. it wrote some bytes successfully out to the socket) it then replies with the number of bytes actually written. If it fails either because the socket has been closed, or the write failed, it logs the reason and replies with a failure message. If the `<no of bytes>` was larger than the number of `<bytes>` it was supplied with then it writes as much as it can.

If successful it replies with:

```
<sender>
socket
writeOK
<no of bytes written>
```

or

```
<sender>
socket
writeFailure
```

otherwise.

5.6.3 The Socket `close` Request

The Socket close request takes the following form:

```
socket
<sender>
close
```

or from the IOP GUI front end:

```
(<sender> close)
```

In response to such a request, if the socket is still open it closes it, and remembers this fact, so subsequent requests will always fail. If the socket is already closed, this like all the other requests will result in a fail reply. Upon closing the actor unregisters with the registry, then exits. The process of unregistering is not instantaneous.

If successful it replies with:

```
<sender>
socket
closeOK
```

or

```
<sender>
socket
closeFailure
```

otherwise.

5.7 The Listener Actor

The Listener Actor knows the listening socket that it is managing. It also knows the number of connections that have been made. A listener actor also knows a client actor `<client>`, the one that requested its creation. It is to this actor that it sends the names of the socket actors it generates per incoming connection. The listener has two threads. One thread monitors the listening socket, the other handles incoming messages.

5.8 The Listener `close` Request

```
listener
<sender>
close
```

or from the IOP GUI front end:

```
(<sender> close)
```

If the listener socket is still open it closes it, and remembers this fact, so subsequent requests will always fail. If the listener is already closed, this will result in a fail reply. After successfully completing this shutdown procedure the actor unregisters with the registry, then exits. The process of unregistering is not instantaneous.

If successful it replies with the message:

```
<sender>
listener
closeOK
```

otherwise it replies with a failure notification:

```
<sender>
listener
closeFailure
```

The listening thread does not accept commands.

All it does is listen on it's port, when a connection is made it creates a new socket actor, whose name will be of the form:

```
"connectionsocket.<listener pid>.<requestNo>"
```

and replies:

```
<client>
listener
newConnection
<connection socket name>
```

5.8.1 Listener Notes

The Listener actor has a signal handler that waits on any child in response to a `SIGCHLD` signal delivery. This prevents the exiting of any spawned actors from remaining in the system as zombies.

5.9 The Executor Actor

The executor actor allows other actors to execute commands in the underlying operating system.

5.9.1 The Executor `executor` Request

An Executor executor request takes the following form:

```
executor
<sender>
<command>
```

or from the IOP GUI front end:

```
(<sender> <command>)
```

In response to such a request, the executor forks off a child process, which using the C routine system() executes the command specified by calling

```
/bin/sh -c <command>
```

Once the system call has ended, the child process responds to the <sender> with the appropriate exit code, as described by the standard C library.

```
<sender>
executeOK
<exit code>
```

Note that because the forked child will share the parent's file descriptor table, any output to stdout by the child will be directed to the registry, and presumably result in confusion. For this reason it is best to design one's commands to be silent. Output to stderr will, like any other actor's error stream, be redirected to the output and error window of the GUI, see section 4 for more details.

5.9.2 Executor Notes

The Executor actor has a signal handler that waits on any child in response to a SIGCHLD signal delivery. This prevents the exiting of any spawned actors from remaining in the system as zombies.

6 The .ioprc File

The .ioprc file, which should be situated in the users home directory, allows for customization of IOP. Here is a sample:

```
/home/iop/SRI/PlethoraOfDemos/input.txt
#this file is /home/iop/.ioprc
#this is a comment
font size = 12
#font style = bold
font type = Lucinda Sans
show font familes = true
window width = 550
window height = 550
start maude iop_maude_wrapper     /usr/local/maude-linux/bin
#This is the location of my maude ^^^^^^^^^^^^^^^^^^^^^^^^^^
#Yours might vary!
#start socketfactory iop_socketfactory *FIFO_IN* *FIFO_OUT*
start graphics2d iop_graphics2d_wrapper /usr/iop
#This is the location of my iop binaries ^^^^^^^^
```

```
#Yours might vary!
#start executor iop_executor
#start filemanager iop_filemanager
#start pvs iop_pvs_wrapper
select maude
```

An IOP started with such a `.ioprc` file will have a square geometry. It will also have as its font twelve point Lucinda Sans. It will include a long list of all fonts avaiable in the system in its initial error window. It will consist of the system, GUI, Maude, and Graphics 2D actors. The currently selected actor in the GUI choice widget will be Maude.

7 Writing and Incorporating New Actors

Incorporating new actors into the system is relatively simple, especially if the new actors themselves do not require the ability to create other new actors. Typical examples of these actors would be new formal reasoning tools. Incorporating new actors that can themselves create other actors requires either following the required protocols necessary for *meta-actor* communication with the registry, see section 3 for a description of the various forms of communication, or using the newer *registration interface* with the system actor.

We will deal with the simple case of actors that do not need to procreate, before covering the more complex case. An new actor will, invariably, be incorporated into the system by a `start` request to the system actor, section 5.1, either directly or at startup in the `.ioprc` file. Consequently, we begin by looking at this step in a little more detail.

A start request to the system actor takes the form:

```
system
<sender>
start
<name> <executable> <argv[1]> ... <argv[N]>
```

In response to such a request, the system first finds a unique new actor name based on `<name>`. If `<name>` is unique as is, then this is the name chosen. Otherwise the addition of the smallest numeric suffix that makes the name unique is chosen. It then creates, and registers with the system, an actor whose executable is named by `<executable>`, whose argument array is `argv`, `argv[0]` is set to be the actor's unique name, call it `nameN`. The creation process involves:

- Creating three FIFOs, one each for standard in, out and error. These FIFOs are created in `/tmp/`, and are called

```
iop_<pid>_<nameN>_IN
iop_<pid>_<nameN>_OUT
iop_<pid>_<nameN>_ERR
```

respectively. Here `<pid>` is the process identifier of the main `iop` process.

- A new process is `fork`ed off, and its standard in, out and error streams are redirected to the corresponding FIFOs.
- The new process then executes

  ```
  execvp(executable, argv);
  exit(EXIT_FAILURE);
  ```

  where `argv` is as described above.
- Finally the new actor is registered with the system. This involves, amongst other things, creating separate threads to monitor both out and error streams of the newly created actor.

As a consequence of this, writing an actor involves paying attention to the name one is christened with, i.e. `argv[0]`, and using the appropriate message format when writing to standard out, namely the transport layer described in section 3. In the transport layer a message consists simply of a line of text representing a number (i.e an integer in base ten), followed by that specified number of bytes. For example in Java this can be achieved using the following library

```
public static void sendActorMsg(OutputStream dest, String body){
    String message = "" + body.length() + "\n" + body;
    try{
        dest.write(message.getBytes("US-ASCII"));
    }catch(Exception e){ IO.err.println(e); }
}
```

routine in the `ActorMsg` of the package `g2d.util`, as described in section 5.3. The new actor will also need to parse incoming input on standard in. This also follows the same format of line of a line of text representing a number, followed by exactly that many bytes. Due to historical reasons the text that follows is enclosed in parentheses, with the parentheses being included in the byte count.

The above describes how the system actor creates an actor. If an actor, other than the system actor, needs to create another actor, the process described above is modified slightly in two places. Firstly, the actor doing the creating must vouch for the uniqueness of the newly created actor's name. Secondly, the system actor must be notified of it's creation, so that messages to and from the new actor can be monitored. This can either be done using the low level meta actor communication, or else by using the newer *registration interface* with the system actor.

The *registration interface* of the system actor involves three new requests: an unique *name* request, an *enrollment* request, and an *unenrollment* request. The unique *name* request allows an actor to obtain, from the system actor, a new unique name for it to use in christening a newly spawned actor. This newly spawned actor can then be registered with the system using an *enrol* request, the request must contain the necessary information for the system to incorporate it into its communication infrastructure. A spawned actor can exit the system by sending the system an *unenroll* request.

7.1 The System `name` request

In order to guarantee that actors in the system have unique names, the system actor provides such a service. A unique name request to the system actor takes the form:

```
system
<sender>
name
<name>
```

or if it is sent from the IOP GUI front end it takes the form:

```
(<sender> name <name>)
```

In response to such a request, the system first finds a unique new actor name based on <name>. If <name> is unique as is, then this is the name chosen. Otherwise the addition of the smallest numeric suffix that makes the name unique is chosen. If nameN is this unique name, then the system actor responds with the message

```
<sender>
system
nameOK <name> nameN <iop pid>
```

where <iop pid> is the unique process identifier of the current iop system. If the request cannot be satisfied, then the system responds with a failure message.

```
<sender>
system
nameFAILED <name>
```

The participating actor is then free to use this name to create a new actor. This involves, amongst other things, making the appropriate FIFOs, redirecting the new processes standard streams to these FIFOs, and informing the new actor of its unique name. The process identifier of the iop process is included in the reply to assist in making sure the necessary FIFOs will be unique to this particular running IOP system. The spawned process can then be registered with the system using the enroll request.

7.2 The System `enroll` request

Once a new actor has been spawned by another actor in the system, it needs to be registered with the system actor, so that, for example, it's out going mail can be handled, and any incoming mail can be forwarded. To register an actor the system needs to know it's unique name (as agreed with the system by a prior name request), its unique process identifier (so it can be shut down at the appropriate time), and the names of the FIFOs (so mail can be handled, and later at shutdown, they can be removed from the file system). An enroll request to the system actor takes the form:

```
system
<sender>
enroll
<name>
<pid>
<in_fifo>
<out_fifo>
<error_fifo>
```

or if it is sent from the IOP GUI front end it takes the form:

```
(<sender> enroll <name> <pid> <in_fifo> <out_fifo> <error_fifo>)
```

If the system actor successfully registers the new actor, it replies with

```
<sender>
system
enrollOK <name>
```

If is unsuccessful it replies with:

```
<sender>
system
enrollFAILED <name>
```

7.3 The System `unenroll` request

When voluntarily exiting the system, it is regarded as polite to notify the system. This allows the system to remove the FIFOs from the file system, and reuse the name if required. This is done via the `unenroll` request. An unenroll request to the system actor takes the form:

```
system
<sender>
unenroll
<name>
```

or if it is sent from the IOP GUI front end it takes the form:

```
(<sender> unenroll <name>)
```

The system actor does not reply directly to the sender, since it may no longer be a going concern. It does send the following to the standard error stream:

```
<sender>
system
unenrollOK <name>
```

If is unsuccessful it sends:

```
<sender>
system
unenrollFAILED <name>
```

so either way, some response will appear in the lower GUI window.

References

[1] I. A. Mason and C. L. Talcott. IOP: The InterOperability Platform & IMaude: An Interactive Extension of Maude. In *International Workshop on Rewriting Logic and its Applications (WRLA 2004)*, Electronic Notes in Theoretical Computer Science. Elsevier Science, 2004.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass., 1986.

[3] `http://www.unix-systems.org`. The Single UNIX Specification Version 3 Homepage.

[4] Ian A. Mason and David Porter and Carolyn Talcott. The `JLambda` Language. Technical Report 05-232, MSCS, University of New England, January 2005. Available at `http://mcs.une.edu.au/~iop/Data/Papers/`.