The IOP Model of Interoperation as a Basis for Developing Interactive Maude Applications

Ian A. Mason

University of New England, Armidale, Australia, 2350.

iam@turing.une.edu.au

&

Carolyn Talcott

SRI International, Menlo Park, California, USA, 94025.

clt@csl.sri.com

January 7, 2006

Abstract

In this paper we describe the IOP model of interoperation and its use in developing interactive applications. Using this as a basis we have developed the IMaude library of modules for defining interactive applications based on the Rewriting Logic language Maude. This is an important step towards making Maude specifications more accessible to non-experts, and thus advancing the possibility for formal executable specifications to have more impact.

The IOP interoperation model is asynchronous message passing following the actor model of distributed computation. The IOP framework provides several built in actors, including an actor that supports interactive visualization using the `JLambda` language.

There are two aspects to developing an interactive tool using the IOP framework: (1) adapting input/output to allow the tool to communicate as an IOP actor; and (2) specifying and programming an appropriate actor-like behavior for the tool. For the Maude actor, (1) is the Maude wrapper that

redirects standard input/output and formats messages; and (2) takes the form of extending a set of Maude modules called IMaude.

This approach has been used in several formal modeling applications based on rewriting logic. We use the Pathway Logic application as a running theme to motivate system design and to illustrate its use

Contents

1 Aims **4**

2 The Architecture **5**

3 The Actors **8**
- 3.1 The System Actor 9
- 3.2 The GUI Actor 9
- 3.3 The Graphics 2D Actor 11
- 3.4 The Maude Actor. 12
- 3.5 Writing and Incorporating New Actors 13

4 Pathway Logic Assistant Requirements **14**

5 Interactive Maude **18**
- 5.1 Requirements and Overview 18
- 5.2 The `LOOP-MODE` Module 20
- 5.3 IMaude Data Types 21
- 5.4 IMaude Behavior 24
- 5.5 Rewriting 25

6 The Pathway Logic Assistant as a Maude actor **29**

7 Related work **36**

8 Conclusions and the Future **37**

1 Aims

In order for formal tools to be more generally useful it is important that the tools can interact with one another via simple, well defined, semantically meaningful communication interfaces. In addition it is important for a formal tool to provide natural user friendly means of interaction.

The Maude system [26, 8] is a high performance system based on rewriting logic with many advanced features. Currently the main means of interacting with Maude is via a command line interpreter. Typically, users that want to connect Maude to other tools or provide alternative display mechanisms, must do something ad hoc, for example with Perl scripts, Tcl/Tk, etc.

The IOP project is aimed at developing an infrastructure for allowing tools to interact and interoperate. It was motivated by the specific aim of making it possible for Maude to communicate with other tools, including other instances of itself, web resources, visualization tools, theorem provers such as PVS [9] and SAL [6], as well as to read and write files, and execute shell commands. The IOP interaction model is that of actors [3, 1] communicating via asynchronous message passing, with the IOP registry serving as local post office. IOP comes with a basic set of actors including a System actor, wrappers that encapsulate Maude and PVS as actors, a Graphics 2D actor, communications actors that support sockets, file system access, and program execution, and a GUI interface to the system, that allows the user to communicate as if an IOP actor. Additional actors can be added quite easily. For meaningful integration of a system as an IOP actor, the tool needs to be adapted to send and receive IOP messages in addition to being integrated into the messaging system with a wrapper. For a system with a programmable 'read-eval-print' loop such as Lisp or Scheme based systems, or Maude, this is straightforward, and such systems can be programmed to have different actor behaviors according to an applications needs. We have developed a set of Maude modules called IMaude (Interactive Maude) that serves as a starting point for defining Maude actor behaviors. IMaude is interactive in the sense that rewrite computations are interleaved with communications with the environment, and IMaude's state persists across communications.

The two systems, IOP and IMaude, combined provides the Maude programmer with a much richer modeling environment, with support for developing visualization and animation of Maude specifications in interesting ways, for exporting Maude modules to other tools (based on other formalisms) for alternative analyses and visualizations, and for developing notions of session state that can be saved and resumed. Using the communication actors as a go-between, the Maude ac-

tor or any other tool adapted to become an IOP actor, can talk to any tool that is capable of interacting via an Internet socket connection or the file system.

Several substantial applications have been based on IMaude: an implementation of the Mobile Maude design [19]; a formal model of goal-based autonomous systems, instantiated to a simple rover [33]; an executable model for Strand Space protocol specifications [21]; and the Pathway Logic Assistant [32]. The latter is the most substantial application and has been an key driving force in the development of both IOP and IMaude.

The IOP manual, binaries for Linux and Mac OS X, and setup instructions are available at [27]. The IMaude code is available at [15].

The structure of this paper is as follows. In § 2 we describe the IOP architecture. In § 3 we describe the basic actors, rules for communication, and explain how to incorporate additional actors.

The Pathway Logic Assistant is the first and most substantial application using IOP, serving as motivation and a test bed for the design and development of the IOP interface and the constituent actors. In §4 we discuss some of the functionality required of the Pathway Logic Assistant and sketch a typical use scenario to motivate some of the design of IMaude. In § 5 we describe IMaude, beginning with a discussion of requirements and overview of the design. The key data structures are described as well as basic rules for interaction. We illustrate how to specify a Maude actor behavior as an extension of IMaude, by describing one of the library modules included in IMaude. In § 6 we continue the discussion of the Pathway Logic Assistant, illustrating how some of the interactions in the scenario are supported by rules for PLA behavior and the evaluation of `JLambda` expressions for interactive graphics. We discuss related work in § 7. We conclude with a discussion of future directions in § 8.

2 The Architecture

IOP's design is based on the actor model of distributed computation [1]. 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 they can 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 is typically 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 [34].

However, not all actors are single processes, some consist of two processes. For example, the actors that correspond to formal reasoning tools such as Maude and PVS, usually consist of two processes: the process running the tool, and a wrapper actor acting as a go-between for the tool and the underlying message system. Similarly the Graphics 2D actor is also a two process actor, one process running the Java virtual machine, and the other a C wrapper process also acting as a go-between.

There is no restriction on the language used to write an actor's script or executable. Some are written in C, some are written in Java, some are written in Perl. One simply chooses the appropriate language for the desired task or function that the actor is supposed to perform. Actors can be single threaded or multi-threaded, each according to its needs. They can even consist of several processes written in different languages. For example, the Graphics 2D actor that provides Maude, and any other actor that wishes it, with a graphical toolkit, is written in Java, and requires a thin C process wrapper to interface with the FIFOs. In § 3.5 we describe the process by which new actors can be incorporated into the system.

Apart from the dynamic pool of actors in the system, IOP consists of three independent processes that interact: the `main` process that creates and configures the system; the registry; and a GUI front end. 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, and any other actors that have been requested. A typical IOP process configuration is shown in figure 1. After startup the `main` acts mainly as a signal handler, ensuring clean and graceful shutdown. The registry keeps track of the current actors, and maintains the lines of communication between these actors. The GUI front end, pictured in figure 2, 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 previously composed messages can be loaded, 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).

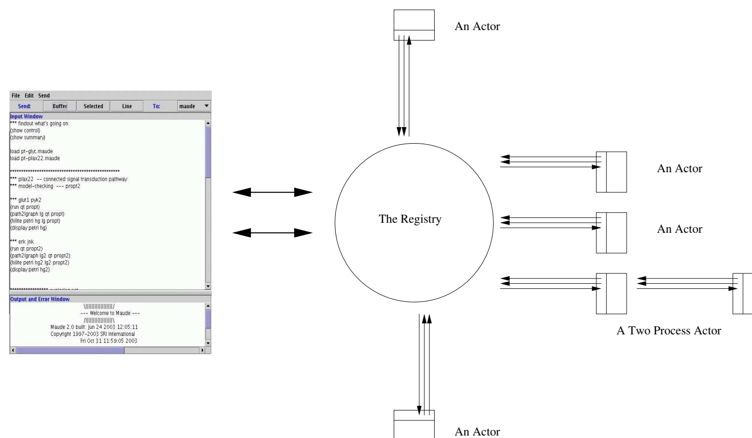

Figure 1: An IOP Process Configuration

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 GUI front end and the registry and other actors. 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
graphics2d
(invoke graph "redisplay")
```

This same message can be sent from the GUI by selecting `Maude` as the destination, and sending the text `(graphics2d (invoke graph "redisplay"))`. Either way the message is transmitted in the transport layer as the sequence of bytes:

```
45\nmaude\ngraphics2d\n(invoke graph "redisplay")\n
```

Simple libraries implement the user layer on top of the transport layer, and allow for reliable cross platform and architecture independent communication. For example in Java this can be achieved using the following static method

```
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); }
}
```

declared in the `ActorMsg` class of the package `g2d.util`, as described in § 3.3.

3 The Actors

The IOP system currently comes with several built-in actors. They are the System actor, the GUI actor, the Graphics 2D actor, the Executor actor, the Filemanager actor, the Socketfactory actor and wrappers to encapsulate Maude and PVS as IOP actors.

The first seven (excluding PVS) 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, any number of instances of each individual actor (other than the System actor) may be explicitly started up by requesting the System actor to do so.

For the purpose of this paper the crucial actors are the System actor, the GUI actor, the Maude actor, and the Graphics 2D actor. The remaining actors, are

described in [17, 28], and no longer play a central role. In fact we expect them to be superseded by the Graphics 2D actor, see § 3.3. We describe the System actor, the GUI actor, the Maude actor, and the Graphics 2D actor each in turn.

3.1 The System Actor

The first major difference between the version of IOP described in the paper [17] and the current version 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 the IOP manual for precise details [28]. 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* provided to make it relatively easy to program actors that spawn new actors. We will discuss the *registration interface* in more detail in § 3.5.

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. Preferring 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.

3.2 The GUI Actor

The role of the GUI front end, depicted in figure 2, is purely as a graphical user interface, allowing the user to interact with any actor in the system. The GUI 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, or the

Figure 2: The IOP GUI

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, or by messaging the System actor.

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; or automatically at startup, by naming the file `input.txt`, and placing it in the directory where you want to execute 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 actor whose name is 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.

3.3 The Graphics 2D Actor

The second major difference between the version of IOP described in the paper [17] and the current version is the stable integration of the Graphics 2D actor. The Graphics 2D actor consists of two intertwined components: A control language, and a class hierarchy.

The control language is an untyped Scheme-like lexically scoped interpreted language, called `JLambda` [14, 20], that provides a runtime interface to the Java class library, as well as some specific classes that make up the second component. The language makes very heavy use of Java's built in reflective capabilities. It is designed to be efficient and expressive enough to enable full and faithful use of any built in Java classes.

The Java class hierarchy, called the `Glyphish` hierarchy [29, 11], is inspired by Joel Bartlett's now deprecated `Ezd` package [4], and Java's 2D [25] implementation, that provides `JLambda` with sufficient built in classes to effectively construct, at runtime, any desired interactive graphical object.

The `Glyphish` hierarchy has been designed both with several benchmark applications in mind, and with architectural generality. The hierarchy makes heavy use of the `JLambda` language infrastructure. Closures [1] in particular, provide a rich language in which to describe control flow, event listeners, and even both `static` and non-`static` methods of dynamically created classes.

The Graphics 2D Actor, here called `graphics2d`, is simply an entry point to the interpreter of the `JLambda` language. Thus the generic request takes the form

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

which simply results in the Graphics 2D actor evaluating the supplied expression in a separate thread of execution. 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

[1] A closure is a lambda expression paired with an environment binding free variables to their values at the point where the lambda was returned as a value.

evaluated. For example if one sends the following two messages to the Graphics 2D actor from the GUI front end

```
(user
  (define respond
      (lambda (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!
```

and will be displayed in the GUI's output and error window. The static method `sendActorMsg` is described in § 2.

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 communication actors largely redundant. Though there is nothing to stop the user from doing this themselves.

3.4 The Maude Actor.

The Maude actor consists of two processes, one running the Maude executable, while the other, called the *wrapper*, acts as an intermediary between Maude and the registry. Any error messages Maude emits are, like all other actor's error messages, redirected to the error and output text area of the GUI front end. Maude's output is interpreted by the wrapper, and then translated to a format acceptable to the underlying *inter-actor* communication system. The process of interpretation consists of replacing symbolic control characters such as `\n`, `\r`, `\t`, `\"`, and `\\` by the appropriate control sequences themselves.

3.5 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 § 2 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. A new actor will, invariably, be incorporated into the system by a `start` request to the System actor, § 3.1, 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 four simple steps. Firstly, one must create three FIFOs, one each for standard in, out and error. These FIFOs are created in `/tmp/`, and are typically 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. Secondly, a new process is `fork`ed off, and it's standard in, out and error streams are redirected to the corresponding FIFOs. Thirdly, the new process then executes

```
execvp(executable, argv);
//error reporting goes here
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 § 2. 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. 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 obtain a unique name for the newly created actor. 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 *enroll* 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 actor an *unenroll* request. For more details on the nature of these three requests the reader is advised to consult the current IOP user manual [28].

4 Pathway Logic Assistant Requirements

Pathway Logic [10, 23, 24, 32] is an approach to modeling biological systems as formally-based executable specifications, using formal methods tools to analyze these models. Specifically, cellular networks—collections of rules describing processes that transmit information (signal transduction) or transform chemicals (metabolism) are modeled using Maude.

A Pathway Logic model consists of a collection of Maude modules specifying the structure and components of a cell; giving rules describing how sig-

nals are propagated in order to control cellular processes such as transcription, metabolism, proliferation, or self-destruction; and defining one or more initial states (called dishes) to study. The Maude modules are organized in four layers: (1) sorts and operations, (2) components, (3) rules, and (4) dishes. The 'sorts and operations' layer can be thought of as the 'logical' signature, declaring the main sorts such as proteins, DNA, and cellular locations, subsort relations and constructors. The components layer specifies specific proteins and other chemicals. The rules layer contains rewrite rules representing biological mechanisms. The dishes layer specifies initial states of interest. An initial state consists of a cell in a supernatant mixture containing signaling ligands. A cell is divided into compartments such as cell membrane, cytoplasm, nuclear membrane, and nucleus. A cell specification determines the proteins and other chemicals of interest contained in each compartment.

For such a model to be useful to biologists, it is crucial to have an interactive visual representation that can be used to navigate and query the model. A few examples of what a biologist might want to do are:

- List the dishes that are available to study.
- Display the network of signaling reactions for a given dish.
- Locate a particular network element, a reaction or reactant.
- Ask for more information about a particular network element.
- Formulate and submit a query about pathways in the network.

The Pathway Logic Assistant (PLA) was designed to meet the requirements listed above and many others. From an architectural point of view, PLA is a collection of IOP actors and external tools. The reasoning engine and driving force of this collection of actors is the IMaude based Maude actor (PLA-M). The user interacts with PLA via the visual representations provided by the Pathway Logic Viewer (PLA-V), and instance of the Graphics2D actor.

To illustrate how a user might explore a Pathway Logic model and to indicate some of the interactions that occur amongst the the actors (and other tools) we sketch a small scenario. The scenario is based on a model of activation of Rac1,[2] a small signaling protein. Rac1 functions as a protein switch that is "on" (activated) when it binds the nucleotide triphosphate GTP (notated `Rac1-GTP`), and "off"

[2]Also known as Ras-related C3 botulinum toxin substrate 1 p21-Rac1.

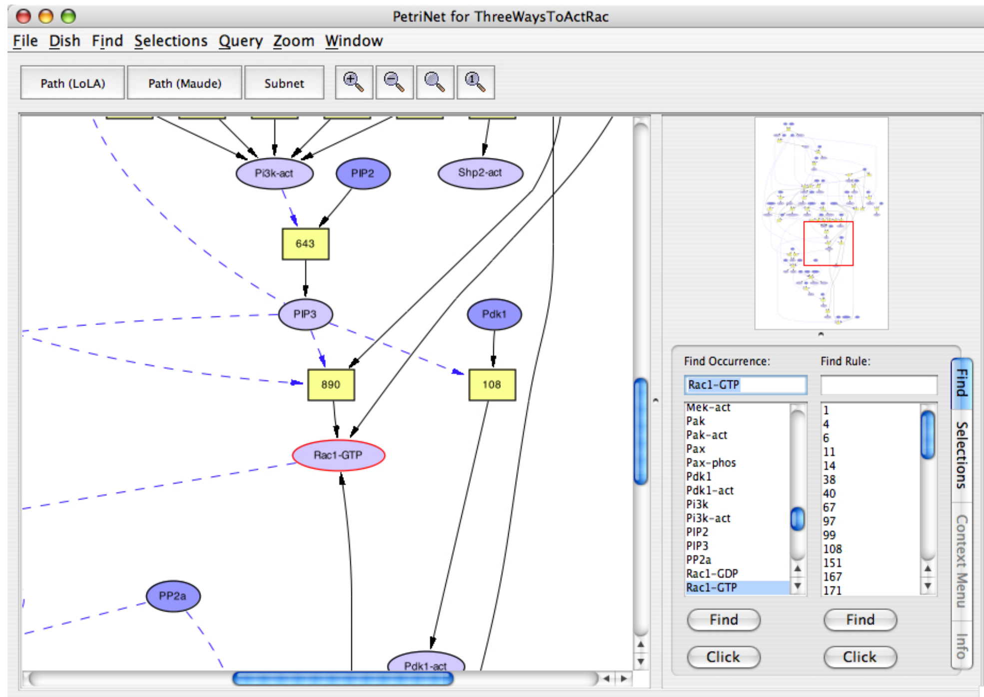

Figure 3: The PLA Viewer.
A screen shot of the PLA Viewer with the model, three ways to activate Rac1, loaded.

when it binds the hydrolysis product GDP, (notated `Rac1-GDP`). The network of signaling reactions for a given dish is represented as a Petri net where places represent signaling components, for example proteins in a particular state and location relative to the cell, and transitions are instances of signaling rules. The scenario starts after the user has asked IOP to create an instance of PLA for a specific model, see figure 3.

In the scenario, `A` $\Rightarrow$ `B` *request* should be read as `A` sends the message *request* to `B`. `PLA-V` is the viewer, and `PLA-M` is the PLA Maude actor.

1. `User` $\Rightarrow$ `PLA-V`: *What dishes are available?* Typically such a request would instigated by a click on dish menu button.

2. `PLA-V` $\Rightarrow$ `PLA-M`: *Send me a list of dish names.*

3. `PLA-M` $\Rightarrow$ `PLA-V`: *The dish names are ...* The message will actually be a

JLambda expression whose evaluation will result in the displaying of a choice menu containing the list.

4. PLA-V ⇒ User: *A choice menu listing dishes.*

5. User ⇒ PLA-V: *Display the network for the dish 3ways2ActRac.* Here the user selects the appropriate item, 3ways2ActRac, in the choice menu.

6. PLA-V ⇒ PLA-M: *Send me the network for 3ways2ActRac.* The action associated with the choice menu results in the sending of this request to PLA-M.

7. PLA-M ⇒ PLA-V: *A graph G representing the network for 3ways2ActRac.* This is achieved by PLA-M computing the network relevant to the dish 3ways2ActRac and its graph representation.

8. PLA-V ⇒ Dot: *Layout G.* The PLA-V actor construct the graph, and then requests that it be annotated with layout information.

9. Dot ⇒ PLA-V: *G with layout.*

10. PLA-V ⇒ User: *An interactive display of G.*

11. User ⇒ PLA-V: *Find the node, N, representing activated Rac1.* This is done by selecting the node, by name, from a list of all node names.

12. PLA-V ⇒ User: The display of G is now centered on N.

13. User ⇒ PLA-V: *Make activation of Rac1 a goal.* This is done by the user clicking on the node N and selecting the goal option. In response to this PLA-V sets an annotation of N to remember its goal status, then (in the next two steps) informs PLA-M, and redisplays the graph.

14. PLA-V ⇒ PLA-M: *Make activated Rac1 a goal in G*

15. PLA-V ⇒ User: The graph is redisplayed with N colored to confirm its goal status.

16. User ⇒ PLA-V: *Find a path to selected goals.* The user selects the find path option from the query menu, or clicks one of the findPath buttons in the toolbar.

17. PLA-V ⇒ PLA-M: *Find a path to the selected goals in G.*

18. PLA-M ⇒ LoLA: *Execute T.* T is a LoLA model checker task produced by PLA-M translating the network and selected goals. Communication with LoLA is done using the underlying filesystem. The language JLambda interpreted by the Graphics 2D actor serves as a go-between here.

19. LoLA ⇒ PLA-M: *A list of transitions achieving the goals.* Using this information PLA-M represents the transition list as a subnet of the current network

20. PLA-M ⇒ PLA-V: *A graph G1 representing the subnet.*

21. PLA-V ⇒ Dot: *Layout G1.*

22. Dot ⇒ PLA-V: *G1 with layout.*

23. PLA-V ⇒ User: *An interactive display of G1.*

5 Interactive Maude

Interactive Maude (IMaude) is a collection of Maude modules that support writing interactive Maude applications. Although, IMaude can be used in Maude alone for simple command line interaction, the intended use is as the basis for specifying the behavior of Maude actors within the IOP framework. In this setting a Maude actor can interact not only with the user, but also with other actors, including actors providing file and socket management services, other Maude actors, actors providing graphical display services, and other formal tools such as model checkers, theorem provers, and so on. Applications are developed by extending the core IMaude system with data structures and rules describing Maude actor behavior specific to the application. IMaude is available from [15]. It comes with several library modules, developed to facilitate interaction with other IOP actors (filemanager, sockets, executor), and a couple of small example applications to play with.

In §5.1 we discuss requirements for Maude actors generalizing the PLA requirements, and give an overview of the IMaude design. In §5.2 we briefly summarize the Maude LOOP=MODE module, which provides the mechanism for interaction. In §5.3 we describe the key data structures used to represent Maude actor state. IMaude extends the data structure modules with three modules for processing input: IMAUDE-STATE, SCHEDULER, and REWRITE. The module IMAUDE-STATE defines the rules for managing initializing, examining and resetting state. It is also discussed in §5.3 The rules for scheduling interactions, defined in the SCHEDULER

module, are described in §5.4. Finally, in §5.5 we describe the `REWRITE` module, treating this as an example of developing a simple Maude actor for interactive rewriting.

We assume the reader has some familiarity with the Maude language [8].

5.1 Requirements and Overview

Implementing an actor behavior requires

1. an interactive loop that maintains state between interactions; and
2. managing asynchronous interactions with other actors.

Requirement 1 is the most problematic as the Maude interpreter is stateless by design (for efficiency). IMaude uses the `LOOP-MODE` module provided by Maude specifically to support writing user interfaces. An alternative would be the use of socket foreign objects that are supported in the most recent Maude release. Although, sockets are somewhat cleaner and more elegant than `LOOP-MODE`, there are two reasons they are not used in the current version of IMaude: sockets were not available when IMaude was developed, and `LOOP-MODE` provides support for parsing and printing Maude terms that is not yet available using sockets. The objective of supporting very general behavior, implies that an IMaude based Maude actor will have many features of an interpreter. Thus additional requirements include

3. a reusable collection of state components that supports a wide range of behaviors
4. extensible data structures for state components along with functions for managing state (initialization, update, retrieval, reset)
5. support for debugging

The need to support asynchronous interaction with multiple actors suggests a need for task management analogous to a virtual machine. Thus a Maude actor's state should include processing state (current and pending tasks), an environment to remember parameters and computed values, and a place to store a log for debugging. IMaude provides two key data types (sorts) to support representation of an actors state: an extensible sort `EVal` to represent data values, and a sort `Request` to represent tasks.

Specifically, an IMaude state has the following components: control; requests; wait4s; environment; and log. The *control* component contains a description of the request currently being processed or the constant `ready`, indicating that no task is currently being processed, Pending tasks are partitioned into two classes: queued requests, stored in the *requests* component; and suspended tasks waiting to handle incoming messages, stored in the *wait4s* component. Wait4 tasks play the role of either call-backs or service listeners. The *environment* component contains a set of entries mapping identifiers to elements of `EVal`. The *log* component is a list of log items. It supports debugging by allowing events and status to be recorded as requests are processed, without interrupting the processing. Additional support for debugging is provided by commands for browsing and resetting state components.

There are two forms of interaction with IMaude: commands and requests. Commands are typically submitted directly by the user and are used to support debugging. Commands are handled upon receipt, and generally result in a reply to the user (printed on the terminal or IOP's output window). Requests maybe submitted by the user, sent in messages by other actors, or generated in the process of handling some other request. Requests are queued and processed when enabled, possibly resulting in messages being sent to other actors.

IMaude provides rules for interpreting commands, rules for dispatching incoming messages, as well as rules for selecting requests from the request queue to process. Use of requests to break processing into small tasks together with the wait4s and requests components of the state provide the support for responsive asynchronous interaction with other actors.

Finally, the IMaude module `REWRITE` defines rules to handle requests for manipulating terms by reducing to canonical form, applying functions or rewrite rules. Since defining requests is the main task in developing an IMaude application, description of this module will also serve to illustrate the extension process.

In the following we describe the data structures making up the IMaude state and the scheduler in more detail. To be self-contained we begin with an introduction to the `LOOP-MODE` module of core Maude.

5.2 The `LOOP-MODE` Module

The `LOOP-MODE` module shown below is the mechanism used to support building user interfaces by providing a basic read-eval-print loop.

```
mod LOOP-MODE is
protecting QID-LIST .
sorts State System .
```

```
op [_,_,_] : QidList State QidList -> System [ctor special(..)] .
endm
```

A `LOOP-MODE` system has the form

```
[inQ,S,outQ]
```

where `inQ` (input queue) and `outQ` (output queue) are lists of quoted identifiers and `S` is the system state. In what follows we will adopt the usual Maude practice and abbreviate *quoted identifier* to *qid*.

The state `S` is constructed from application specific data structures and is rewritten using application specific rules. The state persists between input/output actions until the loop is exited. `inQ` is a stream that receives input directed to the loop from standard input and `outQ` corresponds to a stream connected to standard output. The loop mode reader converts the input byte stream into a qid list using the Maude tokenizer (each qid represents a token), and conversely the output qid list is converted to a byte stream by Maude. Input qid lists can be parsed into Maude data structures using the `metaParse` function and conversely, Maude data structures can be converted to qid lists using the `metaPrettyPrint` function. Thus loop mode can easily function as a traditional *read-eval-print* loop.

To develop a user interface using `LOOP-MODE` one needs to define the `State` data type and rules for processing input from the input stream, possibly modifying the state and generating output (see [8] Chapter 11). In the case of the IMaude actor, IOP messages are received in the input system component and sent by placing them in the output system component.

5.3 IMaude Data Types

As mentioned above, the key sorts used to represent IMaude state are `EVal` and `Request`. `EVal` is essentially a tagged union of sorts, thus making it easy to extend. Injection functions are used to form a tagged union rather than simply making sorts such as `QidList` subsorts of `EVal` to avoid confusion in, and possible collapse of, the sort hierarchy. IMaude defines two `EVal` subsorts: `QVal` and `TermEVal`. Elements of `QVal` are of the form `ql(toks)` where `toks` is a qid list, and `ql` is the injection function, i.e. `QVal` is the image of the sort `QidList`. Elements of `TermEVal` are of the form `tm(modname,term)` where `modname` is a qid naming a module, `term` is the metarepresentation of a term in that module, and `tm` is the injection function. There is also a function `showEVal` that is used to produce printed representations (a qid list) of elements of `EVal`.

`EVal` can be extended by adding new subsorts. To do this an application developer just needs to declare the subsort, and injection function to tag elements and define the `showEVal` function on the new subsort.

An element of the sort `Request` has one of the two forms

`req(reqid,eval,reqQ)` or `creq(reqid,qids,eval,reqQ)`

where `reqid` is a qid identifying the request, `eval` is the parameter, an element of `EVal`, `qids` is a list of qids, and `reqQ` is a list of requests, possibly empty, to be used in determining how to continue when the request is processed. We say a request is serving as a continuation if it appears in a wait4 or in the request list of another request. The `creq` form is used when it is necessary to separate parameters supplied to a request serving as a continuation at continuation time (its `qids` parameter), from the parameter supplied at request creation time (its `eval` parameter). The function `supplyPars(req,toks)` adds the qidlist `toks` to second parameter of the request `req`. In the case of a request of the form `req(reqid,eval,reqQ)` this is only defined if `eval` has the form `ql(qids)`.

An IMaude state has the form

```
st(control,wait4s,requests,environment,log)
```

In the following describe each of the five components.

The *control* component. The *control* component (sort `Control`) of an IMaude state reflects what IMaude is currently doing. An element of `Control` is either the constant `ready` or of the form

```
processing(req)
```

indicating that a request `req` is currently being processed.

The *wait4s* component. The *wait4s* component is a set (sort `Wait4Set`) of elements (sort `Wait4`) of the form

```
wait4(aname,toks,reqQ),
```

where `aname` is the name of an actor from whom IMaude expects a message, and `reqQ` is a list of requests that specifies what IMaude should do when such an expected message arrives. The qid list, `toks`, indicates the reason for waiting, and

is currently just used for debugging purposes. When a message arrives from the named actor, the message tokens are added to the qid list parameter of each request in `reqQ`, using the function `supplyPars`, and the instantiated requests are queued for processing. As mentioned above, an element of the wait4s component can play the role of a call-back, a standard technique used in many programming languages for asynchronous communication, or the role of a server instance listening for input on a connection.

The *requests* component. The *requests* component is a list of requests (sort `RequestQ`) waiting to be scheduled.

The *environment* component. The *environment* component is a set (sort `ESet`) of entries (sort `Entry`) used to store values for later use. An entry has the form

```
e(etype,ids,eval)
```

where `etype`, a quoted identifier, is the entry type, and `ids`, a qid list, identifies the entry within the type, and `eval` is the value to be stored. Together, the pair `etype` and `ids` are expected to uniquely identify the entry.

There are several functions for manipulating entry sets, including:

- `getEntry(es,etype,ids)` is the entry identified by the pair `(etype,ids)` in the entry set `es`, or an error value if there is no such entry.
- `removeEntry(es,etype,ids)` is the entry set obtained by removing the entry in `es` identified by the pair `(etype,ids)`, if any.
- `addEntry(es,etype,ids,eval)` is the entry set obtained by first removing any entries identified by `(etype,ids)` from `es`, and then adding the entry `e(etype,ids,eval)`.

where in the above, `es` has sort `ESet`, `etype` has sort `Qid`, `ids` has sort `QidList`, and `eval` has sort `EVal`.

The *log* component. Finally, the *log* component is a list (sort `Log`) of log items (sort `LogItem`). Each log item has the form

```
log(id,toks,eval).
```

One use of log component is to record status of interactions with other actors, and other use is to record a trace of interactions.

The `IMAUDE-STATE` module provides rules defining commands to initialize, examine, and reset components of the state. The rule `[ini]` initializes the control to `ready` and leaves the remaining components empty.

```
op init : -> System .
***              inQ      ctl   wait4s reqQ  es   log   outQ
rl[ini]: init => [nil, st(ready, none, nil, none, nil), nil] .
```

There are commands to display, via a message to the user, the current value of a state component, as well as commands to reset, to their initial value, each of the different state components. For example

- `(show control)` prints the control component, `(reset control)` resets it to `ready`.
- `(show eset)` prints the environment, `(reset eset)` resets it to `none`.

There are also commands to show or remove specific entries.

5.4 IMaude Behavior

The `SCHEDULER` module provides rules to control the processing of input other than commands. The first qid of the input queue is used to classify the input type as a command, request, or a message expect from a known actor. The boolean function `isReq` is used to determine if a qid is a request identifier. When a new request is defined, an axiom for `isReq` must be added so it will be properly handled.

The following rules handle input other than commands. If the first qid of the input is a request identifier, the input qids are turned into a request that is appended to the request queue. This is handled by the rule `[read.input]`.

```
crl[read.input]:
[token InQ, st(ready,wait4s,reqQ,es,log), OutQ]
=>
[nil,
 st(ready,wait4s,(reqQ,req(token,ql(InQ),nil)),es,log),
 OutQ]
if isReq(token) .
```

where `token` has sort `Qid`.

If the first qid of the input is the name of an actor with a wait4 entry, the wait4 entry is removed, and its requests component is appended to the request queue after supplying the input qids to each request. This is handled by the rule `[schedule.wait4]`.

```
rl[schedule.wait4]:
  [aname InQ,
   st(ready,(wait4(aname,toks,reqQ') wait4s), reqQ, es,log),
   OutQ]
   =>
   [nil,
    st(ready, wait4s,
       (reqQ, supplyPars(reqQ',aname InQ)), es,log),
    OutQ ] .
```

where `aname` has sort `Qid`, and `reqQ'` has sort `RequestQ`.

If there is no pending input, a pending request can be scheduled. The boolean function `enabled` is used to determine if a request is enabled, in the context of a wait4 set. For example interactions with each known actor can be sequentialized by disabling a request that might result in sending a message to an actor for whom there is already a wait4 entry.

The rule `[schedule.request]` applies when there is no pending input. The next request to schedule is determined by evaluating

```
findEnabled(wait4s, reqQ, nil)
```

which returns a pair consisting of the first enabled request in `reqQ`, if any, and the rest of the requests.

```
crl[schedule.request]:
  [nil, st(ready, wait4s, reqQ, es,log), OutQ]
  =>
  [nil, st(processing(?req), wait4s, reqQ', es,log), OutQ]
  if (?req @ reqQ') := findEnabled(wait4s,reqQ,nil) .
```

where `?req` has sort `Request`.

5.5 Rewriting

The REWRITE module defines requests for querying modules loaded into Maude. Terms can be reduced and rewritten using the default interpreter or by specifying a list of rules to apply. The results are saved in the environment for further processing. Also, functions from the object module can be applied to arguments stored in the environment. In all cases the request continuation, with no additional arguments provided, is queued once the environment is updated.

The REWRITE module makes essential use of the Maude META-LEVEL (see [8] Chapter 10), including the meta-representation of terms, and especially the descent functions such as metaParse, metaReduce, metaRewrite, metaApply, and metaPrettyPrint, that provide efficient access to the syntactic and semantic functions of Maude that manipulate terms and modules.

Naming things in the environment. The setqc, letc, and applyc requests provide a means for storing qid lists and terms in the environment. [3]

- (setqc vname qids) adds an entry e('setq, vname, ql(qids)) to the environment.

- (letc vname modname sort <exp>) attempts to parse the qid list that results from reading and tokenizing <exp>, in the module named by modname as an element of sort sort. [4] If successful, the resulting term is reduced to canonical form, res, and an entry

  e('let,vname,tm(modname,res))

  of type let, with identifier vname, and value tm(modname, res) is added to the environment.

- (applyc modname vname fname arg-1 ... arg-n) applies the function named fname to arguments stored (as lets) in arg-1 ... arg-n in the module named by modname, reducing the application to canonical form, res. The result is saved in vname, i.e., an entry of type let, with identifier vname, and value tm(modname, res) is added to the environment.

[3] Corresponding user commands named by omitting the final c are also provided.

[4] From the users point of view, <exp>, appears as the term would if typed to Maude in the context of the named module, while from the IMaude point of view, it appears in the input component of a LOOP-MODE system as a qid list.

As an example we show the code for the `letc` request. It begins with equations specifying that `letc` is a request identifier, and that `letc` requests, with sufficiently many arguments—they must have at least a value name, a module name and a sort—are always enabled. Requests that are ill-formed because they have too few tokens just remain in the request queue.

```
eq isReq('letc) = true .
eq enabled(wait4s,
           req('letc,ql(vname modname sort toks),reqQ'))
    = true .
```

The rule `[letc]` specifies how to process a `letc` request. First the qid list `toks` is parsed using the descent function `metaParse`, and bound to the variable `res?`. If parsing succeeds, `(res? :: ResultPair)`, the result, `getTerm(res?)`, is reduced using the descent function `metaReduce`, and this result is added to the environment using

```
addEntry(es, 'let, vname, tm(modname,t)).
```

If parsing fails then the environment is unchanged.

```
crl[letc]:
 [nil,
  st(processing(req('letc,ql(vname modname sort toks),
                    reqQ')),
     wait4s,reqQ,es,log), OutQ]
  =>
 [nil,
  st(ready,wait4s,(reqQ,reqQ'),es',log),
  OutQ ]
  if res? := metaParse([modname],toks,sort)
  /\
  t := (if (res? :: ResultPair)
        then getTerm(metaReduce([modname],getTerm(res?)))
        else ''0.Qid fi)
  /\
  es' := (if (res? :: ResultPair)
          then addEntry(es, 'let, vname, tm(modname,t))
          else es fi )  .
```

There are a number of alternative for the behavior of `letc` in the case of failure. An ill-formed request could be removed from the request queue and a log item describing the failure could be stored. Either way, using the commands to examine the state, the user could detect the problem. If the request is simply dropped, there would be no record of the ill-formed request attempt. In the case of failure to parse, an error report could be logged, instead of silently failing.

Rewriting terms in the environment. The `rewritec` and `applyrulesc` requests provide a means for rewriting a term stored in the environment and saving the result.

- `(rewritec nat vname [flag])` rewrites the term stored with etype `let` and identifier `vname`, using at most `nat` rewrites (rule applications). The result is stored back in `vname`. If flag is present the Maude *frewrite* strategy is used rather than the default rewrite strategy.

- `(applyrulesc vname rname rids)` tries to apply each rule named in `rids` to the term stored in `vname`. The result is stored `rname`. Rules that don't apply are simply skipped.

Using the Rewrite IMaude application. We conclude the discussion of the `REWRITE` module with a small scenario illustrating its use. For this purpose, we define a small module `DYNAMIC-LIST` that defines a list data structure whose elements can be rewritten.

```
mod DYNAMIC-LIST is
  sort Elt .
  ops a b c   : -> Elt [ctor] .
  rl[ab]: a => b .
  rl[bc]: b => c .
  rl[ca]: c => a .

  sort DList .   subsort Elt < DList .
  op nil : -> DList [ctor] .
  op _;_  : DList DList -> DList [ctor assoc  id: nil] .

  var e : Elt .  var l : DList .
  op tail : DList -> DList .
  eq tail(e ; l) = l .
```

```
    eq tail(l) = nil [owise] .
endm
```

The element sort `Elt` has three members: `a`, `b`, and `c`. These elements can be thought of as three states of a finite state machine, where the rewrite rules define the transitions

$$\texttt{a} \rightarrow \texttt{b} \rightarrow \texttt{c} \rightarrow \texttt{a}$$

After loading the above module, IMaude and the REWRITE module into Maude and initializing the loop state (using the command `loop init .`) we can use the rewrite requests to query the `DYNAMIC-LIST` module.

To begin we define a list to work with using the `letc` request.

```
(letc l0 DYNAMIC-LIST DList  a ; b ; c)
```

This results in an entry `e('letc, 'l0, tm('DYNAMIC-LIST,l0T))` being added to the entry set component of the state, where `l0T` is the meta representation of the list `a ; b ; c`. (For the curious, it is `'_;_['a.Elt,'b.Elt,'c.Elt]`.) Now we can rewrite this list, say for two steps, and examine the result using the `show entry` command.

```
(rewritec 2 l0)
(show entry let l0)
```

The result is `c ; b ; c`, as the default rewrite strategy rewrites the first component twice. We can get the tail of the list using the `applyc` request.

```
(applyc DYNAMIC-LIST l0cdr tail l0)
```

Now the `l0cdr` entry contains (the meta representation of) `b ; c`. The `applyc` request is especially useful if the term you are rewriting is large and you want to rewrite for s few steps, examine a small part of the result, and then continue rewriting.

We can force rewriting of list elements other than the first using the `applyrulesc` request. For example the rule labeled `bc` only applies to the second list element. Thus

```
(applyrules l0 l0  bc ca)
```

results in the list `a ; c ; c`. We can also request that rewriting use the position fair rewrite strategy by adding a flag to the end of the rewrite request.

```
(letc l1 DYNAMIC-LIST DList  a ; b ; c)
(rewritec 3 l1 t)
```

This results in `b ; c ; a` being stored in `l1`, each list element having been rewritten once.

6 The Pathway Logic Assistant as a Maude actor

The Pathway Logic Assistant (PLA) is the first and most substantial application using IOP, serving as motivation and a test bed for the design and development of the IOP interface and the constituent actors. In this section we show how the scenario of § 4 is realized by PLA. As mentioned in § 4, PLA is a collection of IOP actors and other tools that work together to provide interactive visualization and analysis of the PL models. The main players are the Maude actor, `PLA-M`, whose behavior is defined as an extension of IMaude, and `PLA-V` whose behavior is defined by a `JLambda` library interpreted by the Graphics 2D actor.

Because of the restricted nature of the PL rules, we use Petri nets as the basis for visualization and efficient analysis. In particular given a specific initial state, the Maude rules are specialized to rule instances reachable from the initial state and the resulting specialization is transformed to a Petri net. The transistion corresponding to a rule instance has a 'pre-occurrence' corresponding to each component of the rule left hand side and a 'post-occurrence' corresponding to each component of the rule right hand side. An occurrence is analogous to the notion of species used in many biological interaction and reaction data bases. It specifies the underlying compound, for example a protein, its modifications, such as binding with `GDP` or `GTP`, and its location, such as cell membrane or cytoplasm.

Below we show some of the IOP messages corresponding to elements of the scenario of § 4 as well as some of the entries stored by `PLA-M` and fragments of `JLambda` code that define `PLA-V` behavior for the Graphics 2D actor.

We start with the line

6. `PLA-V` ⇒ `PLA-M`: *Send me the network for `3ways2ActRac`.* The action associated with the choice menu results in the sending of this request to `PLA-M`.

which becomes the IOP message

```
(maude graphics2d displayPetri 3ways2ActRac)
```

sent to the `Maude` actor from the `graphics2d` actor. Upon receipt, the PLA-M listener for messages from the Graphics 2D actor queues a `displayPetri` request with dish name parameter `3ways2ActRac`. As an example we show the rule that defines the response to a `displayPetri` request.

```
crl[displayPetri]:
 [nil,
  st(processing(req('displayPetri, ql(dname toks),reqQ')),
     wait4s,reqQ,es,log),
  OutQ]
   =>
 [nil,
  st(ready,
     wait4s,
     (reqQ, req('topetri, ql(dname),
                req('petri2graph,ql(gname dname),
                    req('graph2graphics2d,ql(gname),reqQ')))),
     es',
     log),
  OutQ]
if ctr := getGlobalCounter(es)
/\ es' := incGlobalCounter(es)
/\ gname := qid("graph" + string(ctr)) .
```

Processing such a request rewrites to a sequence of requests, the first is a `topetri` request passing the dish name `dname` as a parameter. This request has contination a `petri2graph` request which is passed the dish name and `gname`, specifying what to name of the created graph. This request has a further continuation, namely a `graph2graphics2d` request with the graph name as a parameter.

To process the `topetri` request, the Maude actor generates the Petri net corresponding to the dish named `3ways2ActRac` (defined in the query layer module) and saves it in the environment as the entry

```
e('petri-net, gl('3ways2ActRac), tm('QQQ,pnetT))
```

where `'QQQ` is the module that composes the parts of a PL model with auxiliary modules defining Petri net and operations for manipulating them, and `pnetT` is a term meta-representing the computed Petri net. The Maude actor then processes

the `petri2graph` request, computing a graph whose nodes are the Petri net occurrences and rules, and whose edges represent the pre- and post- occurrence relationships. The graph is stored as an entry of the form

```
e('petri-graph, gl('graph0), dg(...))
```

where `dg(...)` is the injection of the graph into the sort `DGraphEVal` a subsort of `EVal` introduced to be able to store graphs as entry values. Finally the Maude actor processes the `graph2graphics2d`, which results in a message sent to the Graphics 2D actor requesting it to construct and display an interactive visual representation of the graph.

```
(graphics2d maude plgraphexp)
```

This corresponds to the scenario line

7. `PLA-M` ⇒ `PLA-V`: *A graph* `G` *representing the network for* `3ways2ActRac`. This is achieved by `PLA-M` computing the network relevant to the dish `3ways2ActRac`.

The message `plgraphexp` is a `JLambda` expression that first defines a function `makeGraph` that takes an empty (Java) graph object and adds nodes and edges to construct the specific graph of interest, and then that calls the function `PLgraph` with this function as an argument.

The following shows fragments of the `plgraphexp` expression for the `3ways2ActRac` model.

```
(let ((nodearray (mkarray g2d.graph.IOPNode (int 25)))
  (makeGraph (lambda (graph)
    (seq
      (apply addONode graph nodearray "EGF" "EGF-out" "0"
                      "init" "true")
         ...
      (apply addRNode graph nodearray "1" "1.EgfR.on" "4"
                      "true")
         ...
      (apply addUniEdge graph nodearray (int 0) (int 4)
                         "true")
         ...
   ))))
  (apply PLgraph "graph0" "PetriNet for 3ways2ActRac" ""
                 "false" makeGraph)
 )
```

The function addONode creates an occurrence node, annotates it with a short label for display, such as EGF, a longer label that also specifies location, such as "EGF-out", a unique identifier, such as "0", and a status indicating whether the occurrence is in the initial state, and adds it to the graph. The function addRNode creates a transition/rule node, annotates it with short and long labels (such as "1" or "1.EgfR.on") and a unique identifier (which is "4" for the rule "1.EgfR.on") and adds it to the graph. The function addUniEdge adds a unidirectional edge from its source argument ((int 0)) to its target argument ((int 4)). (The node identifiers are stored as strings, but converted to integers to be able store nodes in an array nodearray, and to find them when making edge links.) The edge from an occurrence to a rule says that the occurrence is a premise of the rule.

The PLgraph function defines the behavior of the graphics2d actor corresponding to the lines 8-10.

8. PLA-V ⇒ Dot: *Layout G.* The PLA-V actor constructs the graph, and then requests that it be annotated with layout information.

9. Dot ⇒ PLA-V: *G with layout.*

10. PLA-V ⇒ User: *An interactive display of G.*

The first argument to PLgraph, "graph0", is the identifier that the Maude and Graphics 2D actors agree to use to refer to this graph. The next two arguments are used as title and subtitle for the display. The argument "false" is a flag indicating whether or not the graph is a subgraph (subgraphs have fewer defined interactions).

When the PLgraph function is applied, a new graph object is created and filled in by giving it to the makeGraph function. Then the graph is exported in dot syntax, dot is invoked to add layout information, the graph is updated with with this information, and now can be displayed in a PL window also created as part of the execution of PLgraph. Figure 3 of § 4 shows the resulting display.

Graph nodes are made interactive by defining event listeners for them. For example an occurrence node can have a listener that pops up a menu offering items such as "Show info" or "Set goal". Each item has an associated action, represented as a closure— a lambda expression closed in an environment binding the free variables of the lambda expression. An action is executed by applying the closure to two arguments: the interacting object, and the event generated by the

user interaction. For example the `"Set goal"` action is defined by the following lambda expression

```
(lambda (me event)
  (seq
    (apply setGoal graph me)
    (sendMessage "maude" (invoke graph "getUid") "setGoal"
                         (getAttr me "nid" "0") "true")
    (invoke view "repaint") ) )
```

Scenario lines 13-15 correspond to calling the `"Set Goal"` action associated with the node labeled `Rac1-GTP`. Evaluation of the expression `(invoke graph "getUid")` returns the name given the graph by the Maude actor when the graph was created.

13. `User` ⇒ `PLA-V`: *Make activation of `Rac1` a goal.* This is done by the user clicking on the node `N` and selecting the goal option. In response to this `PLA-V` sets the annotation of `N` to remember its goal status, informs `PLA-M`, and redisplays, in the next two steps.

14. `PLA-V` ⇒ `PLA-M`: *Make activated `Rac1` a goal in `G`*

15. `PLA-V` ⇒ `User`: The graph is redisplayed with `N` colored to confirm its goal status.

A PLA graph is made interactive by associating actions with the graph as a whole, using tools and menus associated to the graph to provide user access, as shown in Figure 3 of § 4 shows the resulting display. For example requests to find a path are addressed to a graph, rather than a node. Lines 16-17 of the scenario are realized by the "Find LoLA Path" item of the "query" menu, or by by the "findPath (LoLA)" button in the toolbar.

16. `User` ⇒ `PLA-V`: *Find a path to selected goals.* The user selects the find path option from the analysis menu.

17. `PLA-V` ⇒ `PLA-M`: *Find a path to the selected goals in `G`.*

The closure associated to the "Find LoLA Path" item is the `findLolaPathClosure` whose definition (as part of a `let` naming action closures for the graph interactions) is shown below.

```
(findLolaPathClosure
  (lambda (self event)
     (apply sendMessage "maude" (invoke graph "getUID")
                                "displayLolaPath")
)) ; findLolaPathClosure
```

`graph` is bound in the closure environment to the graph for which this closure is an action. Calling this closure results in an IOP message being sent to the Maude actor, from the graph (using the unique identifier specified at the graph creation time), containing the request `displayLolaPath`. The `PLA-M` rule for processing this request corresponds to the scenario described in step 18-19.

18. `PLA-M` ⇒ `LoLA`: *Execute* `T`. `T` is a LoLA model checker task produced by `PLA-M` translating the network and selected goals. Communication with `LoLA` is done using the underlying filesystem. The language `JLambda` interpreted by the Graphics 2D actor serves as a go-between here.

19. `LoLA` ⇒ `PLA-M`: *A list of transitions achieving the goals.* Using this information `PLA-M` represents the transition list as a subnet of the current network

First the Maude actor computes a LoLA representation, `net`, of the Petri net underlying the graph (and specialized to the specific goals), and a LoLA representation, `task`, of the formula corresponding the "find a path reaching these goals" query. The actual interaction with LoLA is accomplished by sending the following message to the Graphics 2D actor

```
(graphics2d maude  (apply lolaRequest net task reqId maude))
```

and adding

```
wait4('graphics2d, 'askLola gname,
       creq('lolaReply,nil,ql(gname pname toks),reqQ'))
```

to the `wait4` set in the IMaude state. The continuation request has parameters `gname`, the name of the graph being analyzed (the specialized petri net and specified goals) and `pname`, the name to be used to store the path returned by LoLA.

Fragments of the `JLambda` definition of the function `lolaRequest` are shown below. This takes the place of building a proper wrapper and making LoLA an IOP actor.

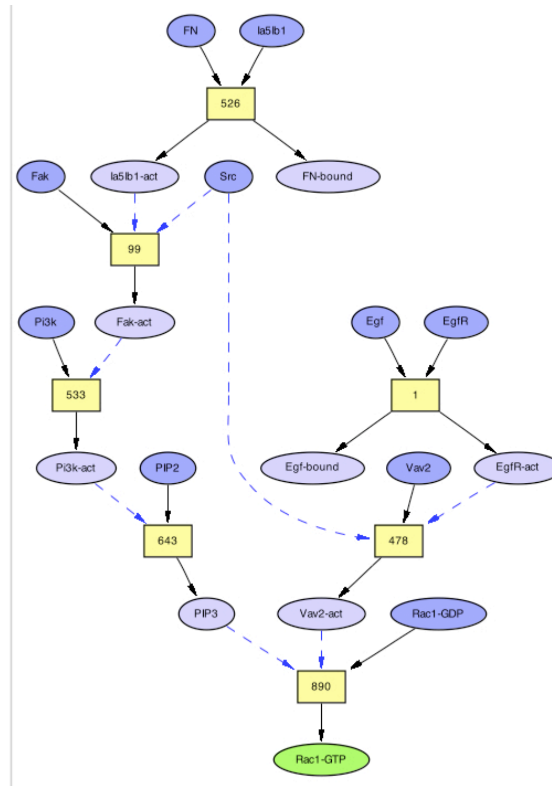

Figure 4: A pathway leading to Rac1 activation.
Ovals colored green are goals, in this case the activated form of Rac1.

```
(define lolaRequest (net task reqId requestor)
   ...
  //write net and task to files for LoLA to read
   ...
   (invoke (sinvoke "java.lang.Runtime" "getRuntime")
           "exec" lolaCommand))
   ...
  // wait for lola to finish
  // then read file written by LoLA call it resPath
   ...
    (apply sendMessage requestor "graphics2d"
```

```
                          (concat retcode " " resPath))
)
```

In the `invoke` form, LoLA is called with command line arguments telling it where to find the net and task input files, and where to write its output file (all contained in `lolaCommand`). When the LoLA process is finished the output file is read, concatenated with the result code and sent to Maude (`requestor`). The message is handled by the `lolaReply` task sitting in the `wait4` set. In particular a Petri net is assembled from the rules used in the path found by LoLA, the corresponding graph is computed and sent to the Graphics 2D actor and displayed in the same way that the top level Petri net is handled. The path for activation of Rac1 found by LoLA is shown in figure 4.

7 Related work

There are two aspects to the IOP and IMaude work. One is moving from a declarative functional language to an *interactive* system while retaining a clean semantics, and the other is *interoperation* of tools. Although we have not emphasized the semantics aspect, we are relying on the basic ideas of interaction semantics for actors [2, 22] to give semantics to Maude actors, without modifying the underlying Maude system. An alternative approach is the idea of Functional Reactive Programming (FRP) [13], where a functional language such as Haskell is extended with constructs such as Monads, Arrows, and I/O to support interaction. The basic Haskell Library can then be extended with primitives for graphics (HGL), robot controllers, and so on. The IOP and IMaude approach is to provide a mechanism for communication with tools or processes providing additional services rather than extending Maude.

The ToolBus [7] is a software coordination architecture. The ToolBus utilizes a scripting language based on process algebra to describe the communication between software tools, providing synchronous and limited broadcast forms of communication. To integrate a tool, an adapter must be written that translate between the internal ToolBus data format and the data format used by the individual tools, and adapts the tool to the ToolBus communication protocols. The IOP coordination model is simply asynchronous message passing taking strings to be the basic communication data. The IOP wrapper for non-interactive tools such as Maude

or PVS is a rudimentary form of adaptor for input/output byte streams. Building on the metalogical expressiveness of Maude, IMaude provides the ability to program coordination scripts as desired. Any other tool or language interpreter with a read-eval-print loop, such as Scheme or Lisp, could be used as well. The real work of interoperation is developing representations that support semantically meaningful exchanges amongst tools. The Pathway Logic Assistant is a good example. Maude representations of Petri nets have been defined to allow Maude to communicate with tools for Petri net analysis, and representations of graphs and `JLambda` expressions have been defined to allow Maude to cooperate with the Graphics 2D actor to support interactive graphical representations of Petri nets.

Several frameworks for interoperation of BioInformatics tools have been developed. The Systems Biology Workbench (SBW) [12], is a modular, broker-based, message-passing framework for communication between applications that aid in research in systems biology, using a common representation syntax, the SBML markup language (`http://www.sbml.org`). SBW comes with a simulator, plotter, adaptors for external simulators, and a generic simulation-control GUI interface. SBW is now part of the BioSpice project, a DARPA sponsored project developing infrastructure for integrative Biology [16, 31]. BioSPICE uses Java Beans component technology for tool integration, and a graphical workflow language to describe flow of data through analyzer tools. BioBike (formerly known as BioLingua)[18, 30] is a programmable knowledge environment that enables interoperation of bioinformatics tools by providing a frame-based representation for diverse data sources and a scripting language (based on Common Lisp) for traversing data and describing different combinations of tool application.

8 Conclusions and the Future

We have described IOP, a communications infrastructure that manages a dynamic collection of actors including: basic communications actors, a Graphics 2D actor, and actors obtained by adapting existing tools to the communication infrastructure. Currently both Maude and PVS have been adapted. We have also described the IMaude set of modules that support defining application specific behaviors for the Maude actor. The IOP-IMaude combination is being used heavily in several ongoing Maude specification projects including the Pathway Logic Project, the Strand Spaces project, and the Formal Checklists project.

Future work includes interoperation with other tools such as SAL [5], improvements and extensions of IMaude, and developing additional `JLambda` li-

braries to support common interactions.

References

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

[2] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[3] H. G. Baker and C Hewitt. Laws for communicating parallel processes. In *IFIP Congress*, pages 987–992. IFIP, August 1977.

[4] J. Bartlett. Ezd – easy-to-use structured graphics for Java. `http://research.compaq.com/wrl/projects/Ezd/home.html`.

[5] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Muñoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, June 2000. NASA Langley Research Center. Proceedings available at `http://shemesh.larc.nasa.gov/fm/Lfm2000/Proc/`.

[6] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Mu noz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, jun 2000. NASA Langley Research Center.

[7] J. A. Bergstra and P. Klint. The discrete time toolbus—a software coordination architecture. *Science of Computer Programming*, 31:205–229, 1998.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott. Maude 2.0 Manual, 2003. `http://maude.csl.sri.com/maude2-manual`.

[9] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial Introduction to PVS. Technical report, SRI International, 1995. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida.

[10] S. Eker, M. Knapp, K. Laderoute, P. Lincoln, and C. Talcott. Pathway Logic: Executable models of biological networks. In *Fourth International Workshop on Rewriting Logic and Its Applications*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[11] B. Funnell. The Glyphics Hierarchy., 2004. http://mcs.une.edu.au/~iop/Data/Papers/.

[12] M. Hucka, A. Finney, H. Sauro, H. Bolouri, J. Doyle, and H. Kitano. The ERATO systems biology workbench: Enabling interaction and exchange between software tools for computational biology. In *Proceedings of the Pacific Symposium on Biocomputing*, 2002.

[13] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, Lecture Notes in Computer Science. Springer-Verlag, 2003. To Appear.

[14] I. A. Mason and D. Porter and C. L. 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/.

[15] IMaude, 2004. http://www.csl.sri.com/~clt/IMaudeWeb/.

[16] S. Kumar. Biospice: A computational infrastructure for integrative biology, 2003. Special issues 7(3-4) of OMICS: A Journal of Integrative Biology.

[17] 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.

[18] J. P. Massar, M. Travers, J. Elhai, and J. Shrager. Biolingua: a programmable knowledge environment for biologists. *Bioinformatics*, 21(2):199–207, 2005.

[19] Mobile Maude, 2004. http://www-formal.stanford.edu/clt/MobileMaude.

[20] D. Porter. An Interpreter for JLambda., 2004. http://mcs.une.edu.au/~iop/Data/Papers/.

[21] Strand Spaces in Maude and PVS, 2004. http://www.csl.sri.com/users/clt/StrandWeb/.

[22] C. L. Talcott. Actor theories in rewriting logic. *Theoretical Computer Science*, 285(2), 2002.

[23] C. L. Talcott, S. Eker, M. Knapp, P. Lincoln, and K. Laderoute. Pathway logic modeling of protein functional domains in signal transduction. In *Proceedings of the Pacific Symposium on Biocomputing*, January 2004.

[24] Carolyn Talcott and David L. Dill. The pathway logic assistant. In Gordin Plotkin, editor, *Third International Workshop on Computational Methods in Systems Biology*, pages 228–239, 2005.

[25] `http://java.sun.com/products/java-media/2D/forDevelopers/java2dfaq.html`. The Java 2D FAQ.

[26] `http://maude.cs.uiuc.edu`. The Maude Homepage.

[27] `http://mcs.une.edu.au/~iop`. The IOP Homepage.

[28] `http://mcs.une.edu.au/~iop/Data/Manual/`. The InterOperability Platform Manual.

[29] `http://mcs.une.edu.au/~iop/GraphicsActor2D/doc/`. The Graphics2D Actor API.

[30] `http://nostoc.stanford.edu/Docs/`. BioBike.

[31] `https://community.biospice.org`. BioSPICE.

[32] `http://www.csl.sri.com/users/clt/PLWeb/`. Pathway Logic, 2004.

[33] `http://www.csl.sri.com/users/denker/remoteAgents/`. Formal checklists for remote agent dependability, 2004.

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