IOP: The InterOperability Platform & IMaude: An Interactive Extension of Maude

Ian A. Mason [a,⋆], and Carolyn L. Talcott [b,⋆⋆]

[a] *University of New England, Armidale, NSW, Australia* iam@turing.une.edu.au

[b] *SRI, Menlo Park, California, USA* clt@csl.sri.com

Abstract

We describe a platform, IOP, for the interoperation of formal reasoning tools, and an adaptation of Maude, IMaude, that utilizes this platform. Three applications of IMaude and IOP to real world problem domains are described.

Key words: Rewriting logic, interoperability, visualization.

1 The 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 [1,2] is a high performance system based on rewriting logic with many advanced features. Currently the 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 interoperate. It was motivated by the specific aim of making it possible for Maude to communicate with other tools, including other instances of itself,

⋆ 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.

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

⋆ ⋆ ⋆ Thanks to the anonymous referees and Mark-Oliver Stehr for helpful comments.

web resources, visualization tools, theorem provers such as PVS, as well as to read and write files, and execute shell commands. The IOP interaction model is that of actors [3,4] communicating via message passing, with the IOP registry serving as local post office. IOP comes with a basic set of actors including a Maude actor, a PVS actor, a Graphics actor, and communications actors that support sockets, file system access, and program execution. Additional actors can be added quite easily. The Maude actor is an *interactive* extension of Maude that we call IMaude. It is interactive in the sense that rewrite computations are interleaved with communications with the environment, and the IMaude's state persists across communications. IOP 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. The reflective capability of Maude makes Maude well suited to programming such interactions and has been crucial in our studies to date. Using the communication actors as a go-between, the Maude actor can talk to any tool that is capable of interacting via an internet socket connection. Although the main actor in the current IOP is the Maude actor, IOP also currently incorporates a PVS actor, and the basic IOP infrastructure is independent of Maude and could be used to endow other tools with communication capability. The IOP manual, binaries for Linux and Mac OS X, and setup instructions are available at `http://mcs.une.edu.au/~iam/IOP/`. The IMaude code is available at `http://www.csl.sri.com/~clt/IMaudeWeb/`

The development of IOP has been largely driven by the particular needs of several substantial applications. In § 2 we briefly sketch three of these applications: the Pathway Logic Workbench, Mobile Maude, and the SCRover. We will then use these examples to motivate the subsequent description of IOP. In § 3 we describe the IOP architecture. In § 4 we describe the basic actors and rules for communication. In § 5 we describe the core set of Maude modules that we use for programming IMaude applications. We conclude with a discussion of future directions.

2 The Applications

The Pathway Logic workbench is the first and most substantial application using IOP, serving as motivation and a testbed for the design and development of the IOP interface and actors. We discuss this application in some detail, and briefly discuss additional features used by two other applications.

2.1 The Pathway Logic Workbench

Pathway logic [5,6,7] is an application of Maude to modeling cellular networks—collections of rules describing processes that transmit information (signal transduction) or transform chemicals (metabolism). Maude rules for network elements have a form equivalent to

```
rl[id]: p1 : l1 ... pk : lk => p1' : l1' ... pm' : lm'
```

where `pi : li` is an *occurrence*, a protein or biochemical `pi` positioned within a cell at the location `li` (membrane, nucleus, endosome ...). Once a set of rules is represented in Maude, the biologist can use the Pathway Logic Assistant to explore the model structure and to ask questions, such as: starting with a cell containing particular proteins and chemicals (in particular locations) can a state be reached matching a particular pattern. These can be answered using execution, search, and model-checking in Maude, or by converting the model to a Petri net and using Petri net analysis tools. The network, subnets, and generated pathways can be visualized using network graphs—graphs with a rule node for each rule, an occurrence node for each occurrence, and edges from left-hand-side occurrences to the rule node and from the rule node to right-hand-side occurrences. An interactive network graph has actions associated either with the graph as a whole or with particular nodes. For example, a rule node has an action that will display the Maude code for the rule. An occurrence node in a subgraph can has an action that extends the subgraph by adding any missing nodes and links associated the occurrence. An extendible graph has action to undo the last extension. These actions allow a biologist to incrementally explore a complex graph.

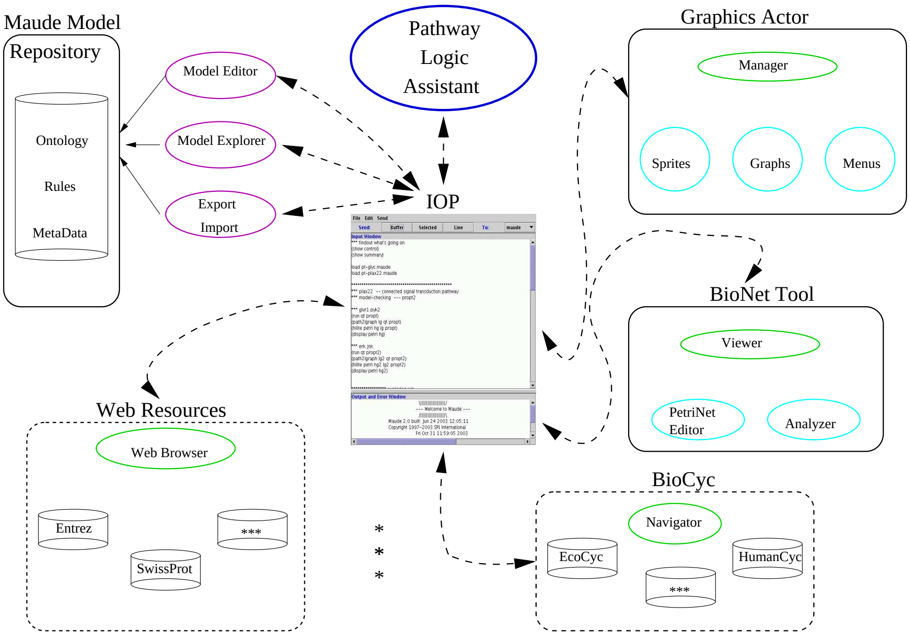


Fig. 1. The Pathway Logic Workbench

A Pathway Logic workbench (figure 1) is being developed to integrate the Maude network models, the Pathway Logic Assistant, and the various auxiliary tools, such as the BioNet Petri net tool, the Dot graph drawing tool, the IOP Graphics actor, and (in the future) Biological databases and web resources, and a GUI for model development. IOP is the underlying infrastructure for the Pathway Logic workbench. We expect that this workbench architecture will be useful for many Maude based applications, adding functionality and enriching the ability to visualize and interact with Maude specifications.

The Pathway Logic Assistant is an IMaude program that serves as command interpreter and coordinator of interactions of Maude and other tools. The IOP communications center (registry) routes requests to appropriate tools, either sending messages directly to IOP actors or using the executor actor to send requests via the underlying operating system and the Filemanager actor to store data to be passed in a file. The Graphics actor is used to display interactive graphs and other visualizations.

2.2 Mobile Maude

To experiment with the use of Sockets and Listeners we built a prototypical mobile object example, using ideas from the Mobile Maude design [8]. Here locations are different Maude actors each with a listener listening on some port for messages. The Maude actor is an IMaude program that can be thought of as an interactive metaobject containing a configuration of possibly mobile objects as part of its state [9]. Each configuration contains a special actor for handling requests to install an object arriving from elsewhere. When there is a message to a remote actor in the base configuration, the metaobject extracts it, opens a connection to the remote listener, and transmits the message. Correspondingly, the receiving metaobject reads the message from its connection, and delivers it to the base configuration. The metaobject only deals with sending and receiving messages, not caring if the message is a normal base-level message or if it transports a mobile object.

2.3 Animating Maude specifications: The SCRover

Providing a visual representation of Maude specifications of distributed systems (system state and evolution) is important to make the specifications meaningful to non-experts, and also to help debug complex specifications.

As a first example, IOP is being used to develop visual, interactive representations our model of the SCRover being developed as part of our NSF-NASA project *Formal Checklists for Autonomous Remote Agents* [10,11]. The objective of the project is to provide higher assurance for software for deep space missions by developing a formal framework for specifying mission goals and the elaboration of goals to goal nets consisting of primitive constraints that correspond to device driver commands to be achieved at specified time intervals.

Rover is visualized using a graphical object that knows its location and orientation. In addition to responding to messages generated by mouse or keyboard input, the rover object can receive messages from other IOP actors.

Thus, like the IMaude actor for mobility, the graphics actor must provide for messages to and from its contained graphical objects. For the initial case study we defined an ad hoc syntax for messages to the rover graphical object and the reactions are hardwired in Java code for the behavior of the rover graphical object (represented as a Java object). Future work is to define a more general syntax for describing active graphical objects. Another task is to define module transformations that automatically add instrumentation for animating object system behavior.

3 The Architecture

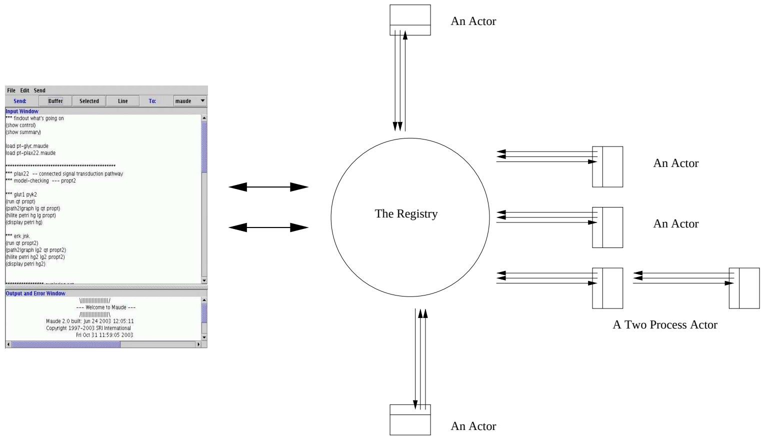
IOP's design is based on the actor model of distributed computation [4]. IOP consists of a pool of actors that interact with one another via asynchronous message passing. The pool of actors is dynamic, it may grow or shrink as time goes by. Actors can be initial actors, created at startup, or be created 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. 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 [12].

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.

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 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.

Apart from the autonomous actors in the system, IOP consists of three independent processes that interact. The `main` that creates and configures the system, the registry, and a GUI front end.



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 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, and message edits can be saved. The lower part displays any output from the actors that isn't inter-actor communication (errors or messages to the user). The `main` and registry are written in ANSI C using pthreads and the UNIX Specification Version 3 [12], while the GUI front end is written using Java's Swing platform. Neither the `main`, nor the registry, nor the GUI are considered as actors in the system, rather they are part of the communication infrastructure. IOP currently runs on Linux, and Mac OS X. Once Maude itself has been ported to the plethora of Windows platforms, a port to these will be constructed, using [13]. The system is designed to be self contained, robust, and extensible. Several different IOP's can run on the same machine without interfering with one another, they can even communicate with one another if the need arises, as it does in the mobile Maude example.

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 Actors

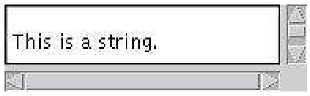
As we have mentioned IOP is configurable, and the wealth of actors in the system depends on the desired application. In the examples discussed in § 2, the important actors are the Maude actor, the Filemanager actor, the Executor actor, the Socketfactory, Listener, and Socket actors, and the Graphics actor. For a detailed description of each actor and the syntax of the message requests they support see [14].

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. So for example the quoted identifier

```
"graphics\nmaude\n(string \"This is a string.\")"
```

would be interpreted by the wrapper as three lines of text, and translated into a form acceptable to the registry, which in turn would interpret it as a message destined to the graphics actor. The graphics actor in turn would interpret it as a request to display the string `"This is a string."` in a window:



The Filemanager Actor.

The Filemanager actor provides any other actor in the system with a uniform way to interact with the underlying file system. It accepts the requests from any actor to read, write, or append to files, and responds with a reply to the requesting actor with a message containing the appropriate information, such as the result of the read, or the success or failure of the write or append.

The Executor Actor.

The Executor actor provides the other actors in the system with the ability to execute any program they care to specify. The Executor actor forks off a child process to execute the requested command, and once the program has finished executing, the child process of the Executor actor replies with the exit code of the requested execution. For example, the message `(user gv foo.ps)` sent from the GUI to the executor actor would result in ghostview window displaying the specified postscript file. The executor is used by the Pathway Logic Assistant to display graphs using the GraphViz toolkit [15].

The Socketfactory, Listener, and Socket Actors.

This trio of actors, or more accurately classes of actors, allows any actor in the IOP system simple structured access to the Internet. It is also our first example of actors creating other actors in response to requests.

Initially, at startup, IOP contains a single Socketfactory actor. A Socketfactory actor responds to two type of requests. It can create a client Socket actor connected to a specified port on a possibly remote host, and if successful replies with the name of the newly created socket actor. Or it can create a Listening socket actor, listening for connections on a particular port, and if successful it replies with the name of newly created listening actor. In both cases, if unsuccessful, the Socketfactory replies with an appropriate failure notification.

A Listener actor is created by a request from a *client actor* to the Socketfactory. The created Listener actor's main task is to listen on the appropriate port, if a connection occurs it creates a new Socket actor for this connection, and notifies it's client actor of the name of this newly created actor. Other than this, the only other thing a Listener can do is close itself in response to a closing request. In response to a closing request the Listener actor closes the underlying operating system listening socket, notifies the registry of its imminent demise, then exits.

A Socket actor is a simple interface to the underlying operating system's socket. An actor can ask a socket actor to write some number of bytes, read some number of bytes, or close itself. In the case of a read request, the number of bytes is taken to be an upper limit. The Socket actor blocks until some bytes are available, it then replies with the number of bytes actually read, and the bytes themselves. If the read fails, the requesting actor is notified. In the case of a write request, the Socket actor attempts to write that number of bytes to the underlying operating system socket.

The Graphics Actor.

The Graphics actor creates graphical objects that the user and other actors in the system can manipulate. Like the Maude actor the Graphics actor is a two process system consisting of a POSIX C wrapper process and a subservient Java process. The Java process is written using Joel Bartlett's Ezd package [16], updated by the first author to Java 2 and Swing, `javax.swing`. Joel's system uses the deprecated Java 1.0 event system, as well as the antiquated `awt` package.

The Graphics actor is a first step towards realizing a graphics algebra with mappings between algebraic data types (as specified in Maude) and graphical objects that have a related structure. Graphical objects are interactive and can be used to interact with related Maude data structures. So far we have defined the following graphical objects: Graphs, Menus, Text, Grids, Containers, and Sprites. These objects are specified using a Lisp style syntax, and messages sent to active graphical objects also have a Lisp style syntax.

As a sample we show a message to the graphics actor requesting creation of a graph as part of a session with the Pathway Logic Assistant.

```
graphics
maude
(graph
 (label qt)
  (nodes
   (node 0 ((label Glucose-out)(shape ellipse)(color lightCyan)))
   (node 1 ((label Glut1-CMin)(shape ellipse)(color lightCyan)))
   (node 2 ((label Glucose-CMin)(shape ellipse)(color lightCyan)))
   (node 3 ((label Glut1-act-CM)(shape ellipse)(color lightCyan)))
   (node 4 ((label 25)(shape box)(color white)
            (onclick "maude\ngraphics\nshow mauderule 25"))))
  (edges
   (edge 0 4 ((color magenta)(label i)))
   (edge 1 4 ((color magenta)(label i)))
   (edge 4 2 ((color green)(label o)))
   (edge 4 3 ((color green)(label o)))))
```

The resulting display is shown in figure 2. The `onclick` node annotations are actions to execute when the user (shift-)clicks on the node. They specify

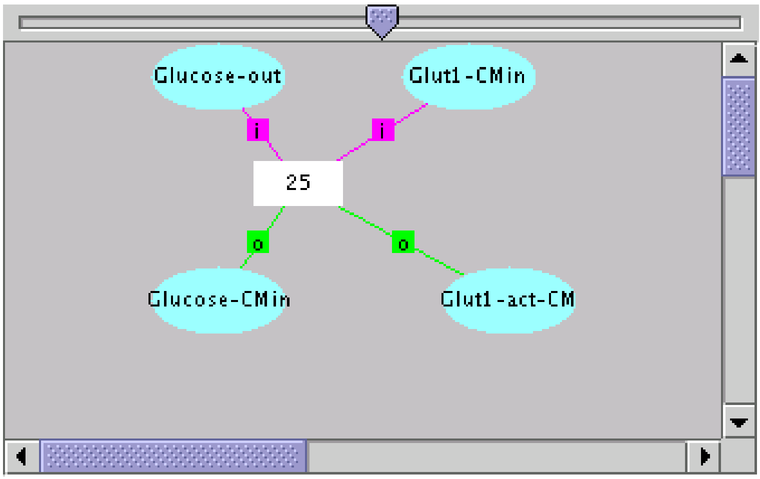


Fig. 2. A pathway graph

a message to be sent by the Graphics actor. Thus (shift-)clicking on the node labeled 23 causes a message to be sent to the Maude actor from the Graphics actor, with request to show the Maude rule 23. Maude replies with a text string which is to be displayed as shown in figure 3.

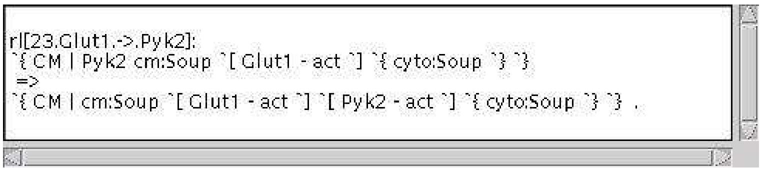


Fig. 3. The result of invoking the show rule action

5 IMaude

IMaude extends Maude to allow interactions with the environment to be interleaved with steps of rewriting. IMaude can send messages to and receive messages from other IOP actors (including the user) or communicate with other systems via files or sockets.

IMaude begins with the `LOOP-MODE` module of core Maude.

```
mod LOOP-MODE is
protecting QID-LIST .
sorts State System .
op [_,_,_] : QidList State QidList -> System [ctor special (...)] .
endm
```

This module is the mechanism used to support building user interfaces by providing a basic read-eval-print loop. A `LOOP-MODE` system has the form `[inQ,S,outQ]` where `inQ` and `outQ` are lists of quoted identifiers (qids) and `S` is the system state that is rewritten using application specific rules provided in a module that includes `LOOP-MODE`. The state persists between input/ouput 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 Maude tokenizer converts the input byte stream into a qid list (each qid represents a token) and conversely the output qid list is converted to a byte stream by Maude.

To develop a user interface using `LOOP-MODE` one needs to define the `State` data type and define rules for processing input from the input stream (see [2] Chapter 11). In the case of core IMaude this is taken care of by the module `INTERACTIVE` and its imported modules, which is the starting point for our IMaude applications.

5.1 IMaude State

In IMaude a `LOOP-MODE` state is a pair consisting of a control component of sort `Control` and a set of entries of sort `ESet`.

```
mod INTERACTIVE is
  inc ENTRY .
  inc CONTROL .
  inc LOOP-MODE .
  op st : Control ESet -> State .
    ... <application independent rules>   ...
endm
```

The application independent rules are discussed below. Entries are used to record results of reductions and rewriting requests for later use. An entry associates an entry value, of sort `EVal`, with a pair consisting of a qid and a qid list. The qid is the entry type, typically the `LOOP-MODE` command that generated the value and the qid list is used to identify the entry amongst those of the same type, often simply the command arguments. Entries (sort `Entry`), with constructor `e` and entry sets (sort `ESet`) are declared (in the module `ENTRY`) as follows.

```
  sorts Entry ESet EVal .
  subsort Entry < ESet .
  op none : -> ESet .
  op _;_ : ESet ESet -> ESet [ctor assoc comm id: none] .
  op e : Qid QidList EVal -> Entry [ctor].
```

The sort `EVal` serves as the union sort for the different values to be recorded. To avoid sort confusion we use conversion functions to map each sort of interest to `EVal`s. The module `ENTRY` defines a number of useful functions for manipulating entries and entry sets, including functions to retrieve an entry from, add an entry to, and remove an entry from an entry set. These functions ensure that an entry set has at most one entry for a given entry type and argument list. In the `LOOP-MODE` setting, printing means producing a qid list to put in the output queue. The function `showESet(es:ESet)` prints the entries of the `es:ESet`, separated by newlines, using the function `showEntry(ee:Entry)`. The IMaude programmer is obligated to provide equations defining

```
showEntry(e(etype:Qid,qs:QidList,ev:EVal))
```

for each entry type used. These are typically defined along with the command (or commands) that create a given entry type as shown below for the `setq` entry type.

The module `ENTRY` provides two subsorts of `EVal`: `TermEVal` and `QEVal`. Elements of sort `TermEVal` are pairs consisting of a qid naming the module, and a `Term` metarepresenting a term of that module.

```
sort TermEVal .
subsort TermEVal < EVal .
op  tm : Qid Term -> TermEVal .
```

Elements of sort `QEVal` are coerced `QidLists`.

```
sort QEVal .
subsort QEVal < EVal .
op  ql : QidList -> QEVal .
```

The control component of an IMaude system state is used to determine when to respond to particular inputs. Inputs can generally be partitioned into requests and replies. Typically a reply is expected in response to a request sent by IMaude. The control component indicates when IMaude is waiting for a reply. Some incoming requests can be handled at any time, others should not be accepted if IMaude is waiting for a reply. At the moment, these requests are lost (the environment should not have sent them). A more robust version might queue them for later consideration. The module `CONTROL` declares the sort `Control`, two constructors, and a function `showControl` that converts a control element into a qid list for output.

```
sort Control .
op ready  : -> Control [ctor] .
op wait4 : Qid QidList -> Control [ctor] .

op showControl : Control -> QidList .
eq showControl(ready) = 'ready .
eq showControl(wait4(mtype:Qid, args:QidList)) =
       ( 'wait4 mtype:Qid args:QidList ) .
```

The `ready` control indicates that IMaude is not waiting for anything. It is the initial value of the control component. The `wait4(id:Qid,why:QidList)` control, as currently used, indicates that IMaude is waiting for a reply from an actor identified by `id:Qid`, where `why:QidList` contains information about the request needing a reply and/or information about what to do when a reply is received. Use of the `wait4` control will be illustrated below.

The module `INTERACTIVE` provides a number of rules for processing application independent commands. The following are some examples.

- `(show control)` prints the system control component, and sends the printed representation to the user.
- `(reset control)` resets the control component to its initial state.
- `(show entry <etype> <args>)` prints the entry matching the type and ar-

gument qids.

- `(remove entry <etype> <args>)` removes the entry completely matching the type and argument qids.
- `(setq <vname> <vids>)` adds an entry `e('setq,<vname>,ql(<vids>))`.
- `(let <vname> <modname> <sortname> <term>)` adds an entry `e('let,<vname>,tm(<modname>,resT))`, where `resT` is the result of meta-parsing and meta-reducing the `<term>` list of qids in the module named by `<modname>`.

As an example we give the rule for processing a `setq` request. We require that a `setq` request only be accepted in a `ready` state, to prevent external modification of the entry set while processing another request.

```
rl[setq]:
['setq vname:Qid InQ, st(ready, es:ESet),  OutQ]
 =>
[nil,
 st(ready, addEntry(es:ESet, 'setq,vname:Qid, ql(InQ))),
 'user '\n 'maude '\n 'setq vname:Qid InQ ]   .
```

Here we use Maude's inline variable declarations. For example `vname:Qid` declares a variable with name `vname` and sort `Qid`. Inline declarations are completely local, and two inline variables are considered the same (within an equation or rule) only if they have the same name and sort. The input component of the system is set to `nil` to indicate that this input has been processed and removed from the input buffer. The contents of the output component will be removed and written to the output stream by Maude as part of the `LOOP-MODE` semantics.

The equation for printing an entry of type `setq` is

```
eq showEntry(e('setq, vname:QidList, ql(qs:QidList))) =
          'setq vname:QidList qid(" = ")  qs:QidList .
```

The operation `qid` converts a string to a quoted identifier. Note that we have reused the variable name `vname` to declare a variable of sort `QidList`. This was done to remind us that although the argument list part of an entry can be any qid list, we expect that it will be a single qid, the variable name, for an entry of type `setq`.

5.2 IMaude applications

To give a flavor of IMaude programming, we give sample rules from the Mobile Maude application and the Pathway Logic application.

Mobile Maude extends the basic IMaude actor behavior with the ability to support communication between its contained object and objects residing in another Mobile Maude actor (or any other actor that understands the message syntax). In particular Mobile Maude listens (using a listener socket) for connections from other actors, reads a message from each new connection, and makes connections to other Mobile Maude actors to send messages. There is a special entry of type `conf` in the Mobile Maude state, initialized at start up, whose value is the metarepresentation of the object configuration it manages. Messages to external objects are extracted from an object configuration using a function `getXmit` and messages from external objects are delivered using a function `deliver`. Both of these functions must be provided as part of the specification of object configurations. In addition the base- and meta-levels must agree on the format of object addresses. For this initial case study addresses are triples of the form `(host,port,local-id)`.

As an example, we show the Mobile Maude rules used to read a message from an external object. We assume that a Listener actor has been created and stored as an entry `e('setq,'listener,ql(lname:Qid))`. When the remote actor opens a connection, the receiver is informed of a new connection with a message containing the name of the Listener actor, the message type `newConnection`, and the name of the Socket actor created for the connection. In response, a read message is sent to the new Socket actor from the Mobile Maude actor. The Mobile Maude actor now waits for the Socket actor to reply, with arguments specifying that the wait is for a read from a listener connection.

```
rl[listener.newCnx]:
[lid:Qid 'newConnection socketName:Qid InQ,
 st(ready, (es:ESet ; e('setq, 'listener, ql(lid:Qid)))),
 OutQ ]
 =>
[nil,
 st(wait4(socketName:Qid, ('listenercnx 'read)),
    (es:ESet ; e('setq, 'listener, ql(lid:Qid)))),
 OutQ  socketName:Qid '\n  'maude '\n  'read '10000 ] .
```

A read reply consists of the Socket actor's name, a read status, and the result of the read `inQ`. A close message is sent to the Socket actor and the Mobile Maude actor waits for a reply, remembering that it is waiting for a close acknowledgement from the Socket actor. The read result is also stored in the `wait4` qid list.

```
rl[listener.readAck]:
[socketName:Qid readAck:Qid InQ,
 st(wait4(socketName:Qid, ('listenercnx 'read)), es:ESet),
 OutQ]
 =>
[nil,
 st(wait4(socketName:Qid,
         ('listenercnx 'close readAck:Qid InQ )), es:ESet),
 OutQ socketName:Qid '\n 'maude '\n 'close ]  .
```

When a close acknowledgement is received, if all is well, that is if

`readAck:Qid == 'readOK`,

the read result with the first element removed (this is the number of bytes read) is put in the input position, as a command for Mobile Maude to interpret, and the control part of the state is set to `ready`. If there is a problem, the user is informed and the read result is discarded.

```
rl[listener.closeAck]:
[socketName:Qid closeAck:Qid InQ,
 st(wait4(socketName:Qid,
         ('listenercnx 'close readAck:Qid toks:QidList )),
    es:ESet),
 OutQ]
=>
(if ((readAck:Qid == 'readOK) and toks:QidList =/= nil)
 then  *** drop the count token
   [rest(toks:QidList), st(ready,es:ESet), OutQ]
 else
   [nil, st(ready,es:ESet), OutQ
   'user '\n 'maude '\n
   'listenercnx readAck:Qid closeAck:Qid toks:QidList '\n ] fi) .
```

A Pathway Logic model is a Maude module that specifies constants and constructors for biochemicals present in cells of interest, and rules describing reactions that are the basic steps of metabolic and signal transduction processes. The Pathway Logic Assistant is an IMaude actor that defines additional data structures and operations to query and transform models, and to visualize models and query results. One of the data structures is a `DGraph`, a structure with nodes and edges each possibly augmented by annotations. Annotations are used to record information that can be used for determining how to render the graph, and what actions are associated with different graph elements. The operation `cseq2dg` produces a `DGraph` from the computation sequence resulting from a query asking for a path leading from an initial state to a state satisfying some desired condition. This is done by implicitly transforming the path into a Petri net-like computation, since there is a natural representation of Petri nets as graphs.

The following rule is the rule in the Pathway Logic Assistant module used to display such a graph. The graph must have been already generated and saved under the name matching `gname:Qid`. It is retrieved using the function `findPetriG`, and the function `dgraph2graphix` prints the graph in a form that can be understood by the Graphics actor, adding annotations for actions and display instructions such as colors and shapes of nodes and edges.

```
crl[display.petri]:
 [ 'display 'petri gname:Qid  InQ, st(ready,es:ESet), OutQ ]
  =>
 (if (pnetG:DGraph == mtDGraph )
  then
    [ nil, st(ready, es:ESet),  OutQ
     'user '\n 'maude '\n
     'display 'petri 'no 'graph  'for gname:Qid InQ ]
  else
    [ nil,
```

```
      st(ready, es:ESet), OutQ
      'graphics '\n 'maude '\n
         metaPrettyPrint(bpMod,
            mkStrConst(dgraph2graphix(pnetG:DGraph))) ]
   fi)
 if  pnetG:DGraph := findPetriG(es:ESet, gname:Qid)   .
```

The function `mkStrConst` converts its string argument into a quoted identifier constant, processing special characters so that `metaPrettyPrint` produces the desired string token. An example message sent to the Graphics actor by an application of the above rule, and its display are shown in Section 4.

6 Related work

There are two aspects to the IOP/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 [17,18] to give semantics to Maude actors, without modifying the underlying Maude system. An alternative approach is the idea of Functional Reactive Programming (FRP) [19], 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 is then extended with primitives for graphics (HGL), robot controllers, and so on. The IOP/IMaude approach is to provide a mechanism for communication with tools or processes providing additional services rather than extending Maude.

The ToolBus [20] 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 a limited broadcast forms of communication. To integrate a tool, an adapters must be written that translate between the internal ToolBus data format and the data format used by the individual tools, and adapts it to the ToolBus communication protocols.

The IOP coordination model is simply asynchronous message passing taking strings to be the basic communication data. Building on the metalogical expressiveness of Maude, IMaude provides the ability to program coordination scripts as desired. The IOP wrapper for non-interactive tools such as Maude or PVS is a rudimentary form of adaptor for input/output byte streams. Some more advanced adaptors have been programmed in IMaude (for example converting representations of graphs). In some cases generic adaptors could be useful, and perhaps we will build on the ToolBus ideas. In the case studies carried out so far, the choice of precisely what external representation to use depends on context, graphs being a good example.

The Systems Biology Workbench (SBW) [21], is a modular, broker-based, message-passing framework for communication between applications that aid in research in systems biology. While Pathway Logic is aimed at qualitative models represented using rewrite rules, the SBW focus is on kinetic models represented using 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. Future work in the Pathway Logic project includes connecting the Pathway Logic Workbench to SBW.

7 Conclusions and The Future

We have described IOP, a communications infrastructure that manages a dynamic collection of actors including: basic communications actors, a Graphics 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 module that support defining application specific behaviors for the Maude actor. IOP is being used heavily in the Pathway Logic Project to develop and experiment with models of biological networks and processes. Its further development will also be motivated by its use in several other current and pending Maude projects.

Ongoing and future work includes systematic development of Graphics actor and the algebra of interactive graphical objects; applications that make use of the interoperation of Maude, PVS and other formal tools; further development of IMaude; and development of an IOP developer toolkit.

References

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

[2] 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.

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

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

[5] 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.

[6] C. 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.

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

[8] Francisco Durán, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Mobile Maude. In *Agent Systems, Mobile Agents, and Applications, ASA/MA 2000*, volume 1882 of *Lecture Notes in Computer Science*, pages 73–85. Springer-Verlag, 2000.

[9] J. Meseguer and C. L. Talcott. Semantic models for distributed object reflection. In *European Conference on Object-Oriented Programming, ECOOP'2002*, volume 2374 of *Lecture Notes in Computer Science*, pages 1–36, 2002. invited paper.

[10] Grit Denker and Carolyn Talcott. Maude specification of the MDS architecture and examples. In *Fifth International Workshop on Rewriting Logic and Its Applications (WRLA'2004)*, 2004. this proceedings.

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

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

[13] http://sources.redhat.com/pthreads-win32/. The Open Source POSIX Threads for Win32 Homepage.

[14] Ian A. Mason and Carolyn C. Talcott. The IOP Manual — http://mcs.une.edu.au/~iam/IOP/.

[15] http://www.research.att.com/sw/tools/graphviz/. The GraphViz Homepage.

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

[17] 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.

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

[19] Paul Hudak, Antony Courtney, Henrik Nilsson, and John 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.

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

[21] 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.