`JLambda`: A Language for Interactive Visualization of Formal Models

Linda Briesemeister[*], Ian A. Mason[†], David Porter[‡], Carolyn L. Talcott[§]

January 7, 2006

Abstract

Recent applications of SRI's formal reasoning tools (including Maude, Sal and PVS) to bioinformatics and cryptography have led to the development of a suite of visualization and interoperation tools. This paper concentrates on the visualization tool `JLambda`, its associated specialized class libraries, and its current uses. Currently, `JLambda`'s primary use is for descriptions of interactive visualizations of complex information, such as Petri nets, graphs, and strand spaces, which are generated automatically by formal reasoning tools such as Maude. `JLambda` has also found a niche in the software development process, as a means of rapidly testing software configurations, and as a "scripting duct tape" that glues test applications together.

Keywords: Formal methods, software engineering, programming languages, visualisation

1 Introduction

Declarative languages with well-defined semantics and logics provide powerful tools for formal executable specification, modeling and analysis of distributed systems. However, complex specifications and analysis results are often difficult to understand. What is needed is language support for interactive visualization of formal models that allows users to navigate and query models based on visual representation of underlying formal structures. Requirements for such a language include a simple syntax that is easily generated and processed by both computer programs and humans, the ability to describe both graphical representations and actions, and dynamic extensibility.

The `JLambda` language is designed to meet these requirements. It is an untyped, lexically scoped, interpreted, Scheme-like language that provides a runtime interface to the Java class library. The interpreter for `JLambda` is written in Java and makes extensive use of Java's built-in reflective capabilities. Following the lead of the Scheme language definition, we require the `JLambda` interpreter to support proper tail recursion. The `JLambda` interpreter's implementation language, Java, however, is not a tail recursive language. In general, choosing a non-tail-recursive language to implement a tail-recursive target language presents obvious difficulties. To overcome these difficulties we first implemented a straight-forward recursive interpreter, and then applied continuation-passing and register-machine transformations to obtain an interpreter capable of properly evaluating any `JLambda` expression. In addition the continuation passing transformation was refined to provide detailed information about the execution state when runtime errors occur, thus making `JLambda` programs easier to debug.

`JLambda` is a core component of the IOP system [1], an infrastructure for allowing formal reasoning tools to interoperate using actor style message passing. In addition to the `JLambda` interpreter, `JLambda` comes with a Java class hierarchy, called the `Glyphish` hierarchy [2], that provides extensible classes to construct interactive graphical objects at runtime. This hierarchy is inspired by Joel Bartlett's now deprecated `Ezd` package [3], and makes substantial use of the Java 2D [4] classes. The hierarchy also utilizes the Scheme-like features of `JLambda`.

[*]SRI International, Menlo Park, California, USA `linda.briesemeister@sri.com`
[†]University of New England, Armidale, Australia, 2350. `iam@turing.une.edu.au`
[‡]University of New England, Armidale, Australia, 2350. `dporter@turing.une.edu.au`
[§]SRI International, Menlo Park, California, USA `clt@cs.stanford.edu`

Closures, λ-expressions together with their lexical evaluation environment, in particular, provide a rich language in which to describe control flow, event listeners, as well as `static` and non-`static` methods of dynamically created or extended classes.

A key objective of the `JLambda` project is a language and implementation that can be used in conjunction with a variety of formal modeling and analysis systems to enable better understanding of complex models and analysis results. `JLambda` is currently being employed in a variety of projects including formal models for autonomous systems, security protocol design and analysis, and development of formal models of biological processes to aid in understanding experimental results, support in silico experiments, and generation of testable hypotheses.

1.1 Related Work

While languages using Java's reflection mechanism or virtual machine are not uncommon, for example there are close to two hundred languages cited in [5] that use Java or the Java Virtual Machine as a basis, roughly twenty of these are classed as Scheme or Lisp like. `JLambda`'s novelty among these is perhaps that it is used by both machines and humans.

We briefly discuss here four examples: SISC [6], Kawa [7], and JScheme [8], and Skij [9, 10]. SISC is a Scheme interpreter implemented in Java whose primary goal is rapid execution of the complete Revised^5 Report on the Algorithmic Language Scheme definition [11], including proper tail recursion and unrestricted first-class continuations. The goal of Kawa is a Scheme environment implemented in Java that compiles Scheme code into the byte-code instructions of the Java Virtual Machine. Kawa provides mechanisms for the definition, creation, and access of Java objects, but does not support tail recursion or first-class continuations. JScheme implements the R4R5 standard except first-class continuations and mutable strings and provides a simple and comprehensive interface to Java via its *Javadot notation* which enables access by name to all methods, constructors, and fields of any Java class. Skij was developed at IBM Watson Labs for exploratory programming in the Java environment. Its purpose is similar to that of `JLambda`, and shares several ideas with `JLambda`. A more detailed discussion of these languages can be found in the `JLambda` manual [12]

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.

2 The Language

This section presents a brief overview of the `JLambda` language. Some examples of `JLambda` code are included to give the flavour of the language. A complete definition of `JLambda` can be found in its reference manual [12].

The `JLambda` language is a minimalistic, call-by-value, left-to-right evaluation order, lexically scoped language with closures. It has exactly the same underlying primitive data types as Java, and access to all of Java's built in packages and classes, as well as any other Java classes found in the class path.

The syntax of the language is based on the usual Lisp notion of an S-expression. An S-expression is either a string of characters or a `List` of zero or more S-expressions. Modeling `JLambda` on Scheme provides it with several well-known advantages of the Scheme language, namely, simple and consistent syntax, plain yet powerful control constructs, and concise, readable programs. In particular, `JLambda` programs are significantly shorter than equivalent Java programs, as is demonstrated in § 2.1. `JLambda` is a small language: it contains only fifty-two keywords, consisting of constructs for: definition and control; arithmetical and boolean operations; constructing and converting primitive types; creating and operating on Java arrays and strings; interacting with Java's class library (creating arbitrary objects, accessing and updating fields, and invoking `static` and non-`static` methods); and throwing and handling Java exceptions. `JLambda` provides no built-in support for general-purpose programming features, such as filesystem access and interprocess control. These features are instead provided indirectly via `JLambda`'s run-time interface to the Java class library.

One of the powerful features of `JLambda` is it's closures. A closure is first class structure that encapsulates a λ-expression (i.e a procedure) together with the lexical environment in which it was created. In `JLambda`, one creates a simple closure that adds a constant to its argument, like so:

```
(define addc (let ((c (int 42))) (lambda (x) (+ x c))))
```

The `addc` closure can then be applied as follows:

```
(apply addc (int 5))
```

which evaluates to the integer `47`.

`JLambda` closures are used heavily in applications as event handlers, and dynamic methods. The following program snippet illustrates the creation and registration of event handlers in `JLambda`. (Note that this excerpt relies on a particular semantics of `JLambda`'s `let` expression: that each binding incrementally augments the lexical environment.) The closure `pressed` responds to mouse-press events by updating the location of a `java.awt.geom.Point2D$Double` object. It is registered as a handler for mouse-press events by invocation of the `setMouseAction` method of the `view` object. We go into more detail about such uses of closures in § 4.

```
(let ((view ...)
      ...
      (prevpoint (object ("java.awt.geom.Point2D$Double")))
      (pressed (lambda (self event)
                 (invoke prevpoint "setLocation"
                         (object ("java.awt.geom.Point2D$Double"
                                  (invoke event "getX")
                                  (invoke event "getY")))))))
  (seq
   (invoke view "setMouseAction"
           java.awt.event.MouseEvent.MOUSE_PRESSED pressed)
   ...))
```

2.1 `JLambda`'s Java Interface

`JLambda` is designed to be expressive enough to enable full and faithful use of any built-in Java classes. To elaborate `JLambda`'s Java interface, we use the following program, which employs Java's AWT API to create a frame and display it in the centre of the screen:

```
(define frameFactory
  (lambda (FrameName)
    (let ((frame (object ("java.awt.Frame" FrameName)))
          (toolkit (sinvoke "java.awt.Toolkit" "getDefaultToolkit"))
          (dim (invoke toolkit "getScreenSize"))
          (h (lookup dim "height"))
          (w (lookup dim "width")))
      (seq
        (invoke frame "setSize" (/ w (int 2)) (/ h (int 2)))
        (invoke frame "setLocation" (/ w (int 4)) (/ h (int 4)))
        (invoke frame "setVisible" (boolean true))
        frame))))

(apply frameFactory "A Frame")
```

In a `JLambda` program any Java class may be accessed by providing its full name, for example, `"java.awt.Frame"`. Arbitrary Java objects are constructed using the `object` form, whose first argument should evaluate to a string representing a Java class. The interpreter then attempts to find a constructor for that class with matching arguments, and uses that constructor and the remaining arguments to construct the appropriate object. If no matching constructor is found an exception is thrown. In the program above, the expression

```
(object ("java.awt.Frame" FrameName))
```

causes the interpreter to invoke the `java.awt.Frame` constructor, with the value of `FrameName` as the constructor's argument. Static non static methods of an object may be invoked using the `invoke` form. To invoke a `static` method via the class rather than the object, one uses the `sinvoke` form. The `lookup` form provides access to static and non-static fields of an object.

3 The Interpreter

We provide in this section an overview of the the design of the `JLambda` interpreter; further details of the interpreter's design and implementation can be found in David Porter's Honours thesis [14]. We begin by showing how the `JLambda` interpreter uses Java's reflection API to enable `JLambda` programs to access, at run-time, the Java class library. We then go on to explain the general design of the interpreter, highlighting the way in which the continuation-passing and register-machine transformations are used.

3.1 Reflection

The Java reflection API is both quirky and low-level. It provides objects that represent meta-level concepts such as classes, interfaces, fields, methods, constructors, and class member modifiers. It also allows for: the ability to access or update the value of an object's field; to invoke an object's method on a given set of values, or construct a new object via calling a constructor on a given set of values. What it *does not provide* is, given a certain sequence of arguments, a means for resolving which method or constructor one should use. Consequently, it falls to the `JLambda` interpreter to decide how this should be done. In Java, method and constructor resolution uses both run-time information and static compile-time information. The run-time type of the target object is used, together with the compile-time types of the arguments. In an interpreted language we have no static type information to rely on; consequently, we must attempt to do the best we can with the possibly *incomplete* information we have at hand. Namely, the run-time types of the arguments.

To evaluate a constructor call after evaluating, and determining the types of the arguments, the interpreter searches for a constructor whose argument types are an exact match. If an exact match is not found, then it searches for the best match among the publicly declared constructors. The notion of *best match* is taken to mean the *least* when taking into consideration widening, the interface hierarchy, and the class hierarchy. There may be many such choices, and, in the current implementation, the interpreter simply chooses one of them, making no effort to resolve ambiguities. If no constructor is found, the expression generates an exception. Resolution of method invocation follows a similar pattern, except that the notion of best is elaborated slightly from the constructor case to take into account the declared return type of the method. The method's return type is examined only when the parameter types match exactly, and in this case the more specific return type is preferred.

Compared to other implementations of interpreted languages that use Java's Reflection API, we have adopted a rather conservative approach. This is because our aim is to provide an interface to Java that is as *simple, faithful, and precise* as possible in an untyped, interpreted language. A rather bolder scheme is outlined by Michael Travers in [9, 15], and used in his language Skij [10], as well as Peter Norvig's JScheme [8].

A small complication to the procedure for method resolution outlined here arises as a result of a long-standing and unresolved bug in the Java Reflection API [16]. The bug has to do with access restrictions on inner class objects, and means that some methods that should be accessible are not. For example, instances of `java.util.Iterator` returned by `java.util.Collection` instances are unusable. When such situations arise we attempt to circumvent the problem by using the `setAccessible` method of the `java.lang.reflect.AccessibleObject` class to suppress the Java language access checking.

3.2 The Interpreter Design

The interpreter for the `JLambda` language consists of three components: a parser, a syntax analysis phase, and an evaluation phase. Choosing Java as the interpreter's implementation language leads to certain complications in the interpreter's design. In general, the simplest way to implement an interpreter of a Scheme-like language is by using a simple recursive evaluation model in which an expression is evaluated by recursively evaluating each subexpression. If such an interpreter is implemented in Java it will exhibit recursive execution behaviour, since it inherits the control structure of the underlying Java system. However, the lack of proper tail recursion in Java means the interpreter overflows the JVM stack when attempting to evaluate arbitrarily long recursions, such as the computation of long lists.

The goal, therefore, was to design the `JLambda` interpreter so that it uses an iterative execution process; this was achieved by implementing the interpreter as a register machine interpreter. In this design, the procedure-calling and argument-passing mechanisms used in the evaluation process are implemented in terms of operations on registers. We thus obtained an explicit-control interpreter that exhibits iterative execution behaviour.

Converting the interpreter design from a simple recursive evaluation model to a register machine interpreter involved two steps. First, we ensured all recursive calls were tail calls, by transforming the interpreter into continuation passing style [17]. If the interpreter implementation language was properly tail recursive, this transformation would have been sufficient to achieve an interpreter with iterative execution behaviour, since properly tail recursive languages guarantee that tail recursion is equivalent to iteration. However, since Java is the implementation language a further step is required, in which we manually transformed tail recursion into iteration.

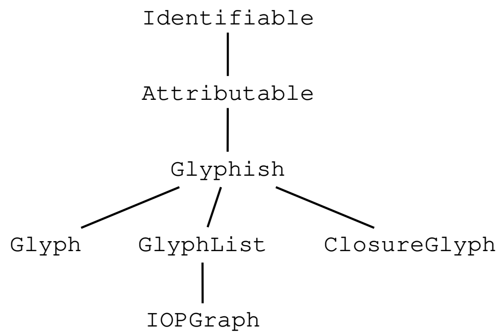
The second step consisted of the transformation of the interpreter from a continuation passing style into a register-based imperative style. This transformation is based on the following observation: if a set of methods call each other only by tail calls, we can first rewrite the calls to use variable assignment instead of argument-passing, and we can then replace method calls with jumps. The register machine transformation consists of systematically performing such rewrites. The result of performing the continuation passing transformation and the register machine transformation was an interpreter with iterative execution behaviour. This interpreter can therefore properly evaluate any `JLambda` expression. The transformation from recursive to iterative interpreter is a standard technique in the programming language community. An example worked out in detail can be found in Felleisen's thesis [18].

The syntax analysis phase serves to improve the interpreter's execution speed. In this phase all lexical variables in a `JLambda` program are replaced by their corresponding lexical addresses in the program structure. Such a program representation enables the evaluator to retrieve variable values directly from known addresses in the run-time environment. Implementing variable lookup operations in this way is a considerable improvement over lookup operations based on searching the environment. Consequently, the addition to the interpreter of a syntax analysis phase yields a significant increase in its execution speed.

The use of continuations in the interpreter facilitates detailed error reporting, thus simplifying debugging. Each continuation possesses an `inform` method that, when invoked, produces an informative description of the execution context that it represents (including filenames and line numbers). When an error occurs an exception is generated. This exception then propagates up the continuation stack accumulating this information, as a *backtrace*. At the toplevel the backtrace can be printed out in various customizable levels of detail.

4 Visualisation Classes

The `JLambda` language evolved at the same time as the `Glyphish` hierarchy and influenced its design, in particular it's use of the `JLambda` closures as event handlers and dynamic methods. This class hierarchy belongs to the `g2d.glyph` package, whose class structure is shown below:



At the root is the `Identifiable` class, instances of which have unique global names, these names are used by the formal tools to refer and interact with them. Beneath this lies the `Attributable` class, an instance of which may have new fields added dynamically, in the form of attributes. An attribute has a name and a value. Attribute values are arbitrary objects, including closures. An attribute with a closure value corresponds to a dynamic method. Directly beneath `Attributable` is the root class of all things *glyph-like*: the abstract class `Glyphish`. There are three related but distinct aspects to the `Glyphish` class, how a `Glyphish` instance: depicts or portrays itself graphically; handles input events from the keyboard and mouse; and positions or transforms itself.

In Joel Bartlett's `Ezd` package a `Glyph` was something that knew how to draw itself, typically as a sequence of shapes, and was capable of accepting and responding to user inputs. The `Ezd` package was developed using the early Java 1.0 event model, and relied on the `java.awt` package as it was then, in Java 1.0.

We adopt a similar, though distinct, conceptual approach. Our approach has been strongly influenced by the newer Java event model, and the clean two dimensional graphics supplied by the Java 2D API. In particular we make heavy use of the `AffineTransform` class of the `geom` subpackage of `java.awt`, and the newer 2D implementations of the `Shape` interface, also of the `java.awt` package. We have also taken advantage of the `Closure` class provided by the `JLambda` language.

The abstract `Glyphish` class has three *main* concrete subclasses: the `Glyph` class, the `GlyphList` class, and the `ClosureGlyph` class. We also single out the specialized subclass of `GlyphList`, `IOPGraph` which we will discuss later. The `Glyph` class is the simplest of the direct subclasses of `Glyphish`. A `Glyph` instance has a single `java.awt.Shape`, border colour, fill colour, and stroke width. A `GlyphList` is a composite, it consists of an ordered list of `Glyphish` things. A `ClosureGlyph` is the most dynamic, it requires `JLambda` closures to implement all the `abstract` methods required by the `Glyphish` API. It provides, in essence, a way of defining, at runtime, `Glyphish` instances whose methods are defined at runtime, rather than compile time.

A concrete `Glyphish` instance portrays itself by implementing the abstract method `paint` declared in the `Glyphish` class. In the case of a `Glyph` instance it will draw itself according to its `java.awt.Shape` field, fill colour, border colour and stroke. In the case of a `GlyphList` it merely delegates, in order, to all of the `Glyphish` elements in its list. The `ClosureGlyph` responds by applying it's private `paintClosure` field to the appropriate `Graphics2D` object of the `java.awt` package.

`Glyphish` instances are capable of handling any input events, i.e. instances of the `InputEvent` class of the `java.awt.event` package. The `Glyphish` class implements each of the `Input` event listener interfaces for the mouse and keyboard. They do so in a uniform way. For each method in the listener interface a `Glyphish` instance has a `Closure` object associated with it. For example in the case of the `mouseClicked` method of the `MouseListener` class, the `Glyphish` class has the private `Closure` field `mouseClickedAction`. This closure will have arity 2, and uses Luca Cardelli's trick of having a self argument to implement Java's `this` pointer. Calling the `mouseClicked` method would result in the `clickedAction` closure being applied to the `this` pointer of the `Glyphish` instance that is responding to the `MouseEvent`, and the event instance itself.

Positioning, moving, and animating `Glyphish` instances is done by applying affine transformations (e.g. translating, rotating, shearing, and scaling) to them.

Another use of the `Closure` class provided by the `JLambda` language is in creating event handlers. For each Java listener class, we provide a corresponding template class that implements the desired listener interface, and extends the `Attributable` class. An instance of the template class can be specified by providing `Closures` for each of the required methods. As mentioned above, we have adopted the convention that these closures take two arguments, the *self* parameter, followed by the *event* parameter.

The specialized subclass `IOPGraph` of the `GlyphList` class is for displaying large complex graphs, that commonly arise in formal models of complex systems. The graphs are rendered in an informative way by using the graph layout tool `dot` [19], and nodes and edges of the graph, being `Glyphish` entities, can respond to events from the user.

5 Applications

`JLambda` is currently being used in several formal modeling and analysis projects. We discuss highlights of three of these projects below.

Typical use of `JLambda` in combination with a formal specification environment involves defining functionality on both sides along with a communication protocol to coordinate the roles of each tool. On the `JLambda` side one defines a library of `JLambda` functions that

- create graphical elements,

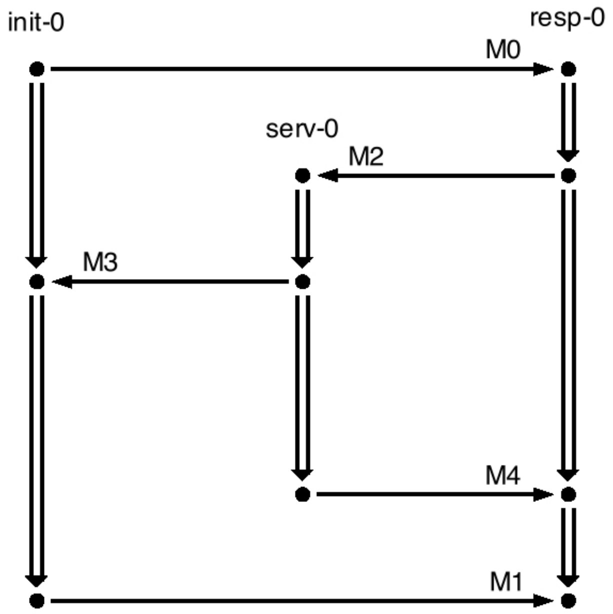


Figure 1: Example display of a strand space bundle

- create a frame with appropriate menus, tools, views,
- define actions associated with user gestures,
- and compose messages for peer tools.

In the projects described below the formal tools include the Maude specification and analysis environment. The IOP platform is used to support communication amongst tools suitably wrapped to behave as actors interacting via asynchronous message passing. IOP also provides a `JLambda` interpreter in the form of the Graphics 2D actor. The interactive Maude module, IMaude, is the basis for defining actor behavior for Maude [1]. The PLA architecture shown in figure 6 is a typical example of the use of `JLambda`.

Executable Strand Spaces Specifications

Strand spaces [20] is a mathematical model for analysis of cryptographic and other protocols. In an ongoing project, Maude and PVS [21] are being used to develop a tool for interactive design and verifiable analysis of security protocols based on Strand spaces. After specifying a protocol (as a sequence of message exchanges) a protocol designer user can specify different initial situations, and use the Maude specification of Strand space protocol execution to execute single runs or search for all possible runs. The result of an execution is a partial order of message send/receive events (called a *bundle*). `JLambda` is used to display this diagram in the traditional Strand space style.
Figure 1 shows an example, while figure 2 is the label key for the figure. Black filled nodes represent events, the double arrows give the ordering of events on a particular strand (actions of a protocol participant), and the single arrows represent the causal ordering between the send of a message and its receive. The `JLambda` library for Strands includes functions to display strand headers (label and initial node), strand segments (double down arrow followed by node), and cross edges connecting corresponding send and receive nodes. Each function takes parameters specifying relative position computed by a bundle layout algorithm executed by Maude. A first version of executable strands is available at [22].

M0 = a, na0

M1 = {nb0}k0

M2 = b, {a, na0, nb0}sh(b,s)

M3 = {b, k0, na0, nb0}sh(a,s)

M4 = {a, k0}sh(b,s)

Figure 2: Key for figure 1

Animating Maude specifications: Goal-based autonomous systems

Providing a visual representation of formal 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 and understand emerging behavior. `JLambda` has been used extensively in a student research project to develop visual, interactive representations of specifications based on the MDS framework for goal-based autonomous space systems [23, 24]. This was part of an NSF-NASA project, *Formal Checklists for Autonomous Remote Agents* [25, 26]. The starting point is a formal executable specification of a goal based system for controlling a simple autonomous rover driving on a grid with obstacles. Goals are constraints on values of system state variables. High-level goals are elaborated into timed constraint nets which are then executed by a scheduler. A timed constraint net is a directed graph whose nodes are time points and whose edges are constraints either on a state variable, or the time interval between the connected points. A goal specification is analyzed by composing it with a formal specification of the device behavior and carrying out formal checks, including execution of a possible run, search for all possible runs, and model-checking temporal properties.

The student project developed a visual representation of the device state along with a dashboard representation of key variables from both the device and the software points of view. The Maude device and goal specifications were instrumented to update the visual display, by sending messages to the Graphics 2D actor, when changes occur. In addition a separate control panel was implemented to allow the user to create and execute a goal net specification of rover tasks without directly typing at Maude. Figure 3 shows the control panel.

Figure 3: The goal net control panel

Figure 4 shows an intermediate stage in the execution of a goal net created using the control panel. The black elements on the grid in the center represent obstacles, the blue triangle represents the robot, and the pink rectangle represents an arm of the robot. The lower panel shows the goal net, with elements color coded to indicate their execution state. Goals are green when they are ready to execute, but have not yet begun execution. They are purple during goal execution, gray when the goal has finished successfully, and orange when the goal has failed. Time points are red before the scheduler fires them and yellow when they have been fired and assigned a time value. Time constraints are blue and are labeled with an interval constraining the time elapsed between the firing of its starting and ending time point.

The `JLambda` library developed for goal-based operation of a grid Rover includes a function for making the grid, given dimensions, initial location of the rover, and locations of obstacles. Grid squares and obstacles are simple glyphs while the rover is a glyph list with components for the rover body, arm, and battery. The IOPGraph class is extended to a virtual Goalnet class by defining attributes enumerating the different kinds of node (time point, goal,

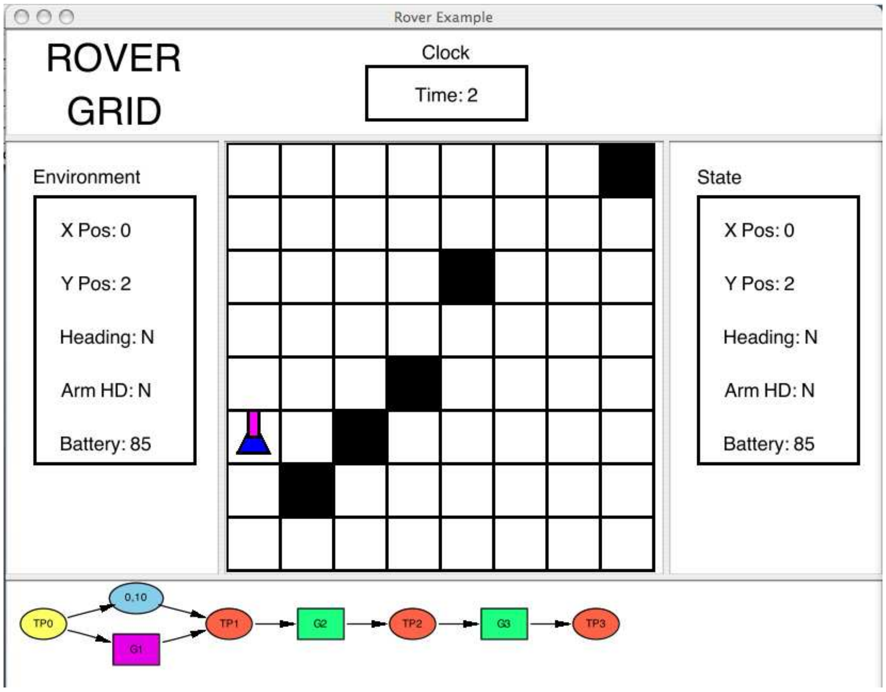


Figure 4: A partially executed goal net

time constraint) and methods (attributes with closure values) to add, remove, and retrieve nodes of each kind. Goalnet nodes are IOPNodes extended with attributes representing key properties. For example a time constraint node has attributes giving the min and max elements of the corresponding specification data structure, and goal nodes have an attribute giving the name of the constrained state variable. Action functions are defined for nodes to display additional information about the node when the user clicks on that node. Each button on the control panel has an associated action defined by a `JLambda` function. Some of the actions update the local goal net, and some of them send update information to Maude to synchronize the Maude goal net state with the visual representation. The project results including code, documentation, and instructions for using the system are available at [26].

Pathway Logic Assistant

Pathway Logic [27, 28, 29] is an application of formal methods to the modeling of biological entities and processes. Currently, we focus on modeling and analysing signal transduction networks in mammalian cells. We represent biological knowledge as formal rules and equations using the rewriting logic Maude. Having a formal model of the biological process in Maude allows for analysis such as dynamically generating pathways using search and model-checking, and transforming rule executions to Petri nets for visualization and further analysis.

The Pathway Logic Assistant (PLA) is the software suite implementing this approach. Figure 5 shows a screen shot of PLA's graphical user interface depicting a signal transduction network as a petri net. In the bipartite graph of the Petri net, round nodes denote places that encode proteins and complexes together with their chemical modification (e.g. an activated Mekk4 is labeled "Mekk4-act"). The transitions between places correspond to rewriting rules in Maude's knowledge base. The places are generally colored in light and dark blue, where the latter indicates that these proteins are part of the starting state used to build the pathways from the Maude knowledge base. In the screen shot, two places are colored green ("Mkk3-act" and "Mkk4-act") and as such marked as goals. One can select proteins as goals or as to be avoided and then search for paths and subnets that avoid the respective proteins while reaching the specified goals. In the top right corner, a thumb nail view of the complete network provides an overview and accepts mouse clicks and drags to move the porthole into the bigger view of the graph on the left. Other features of the GUI

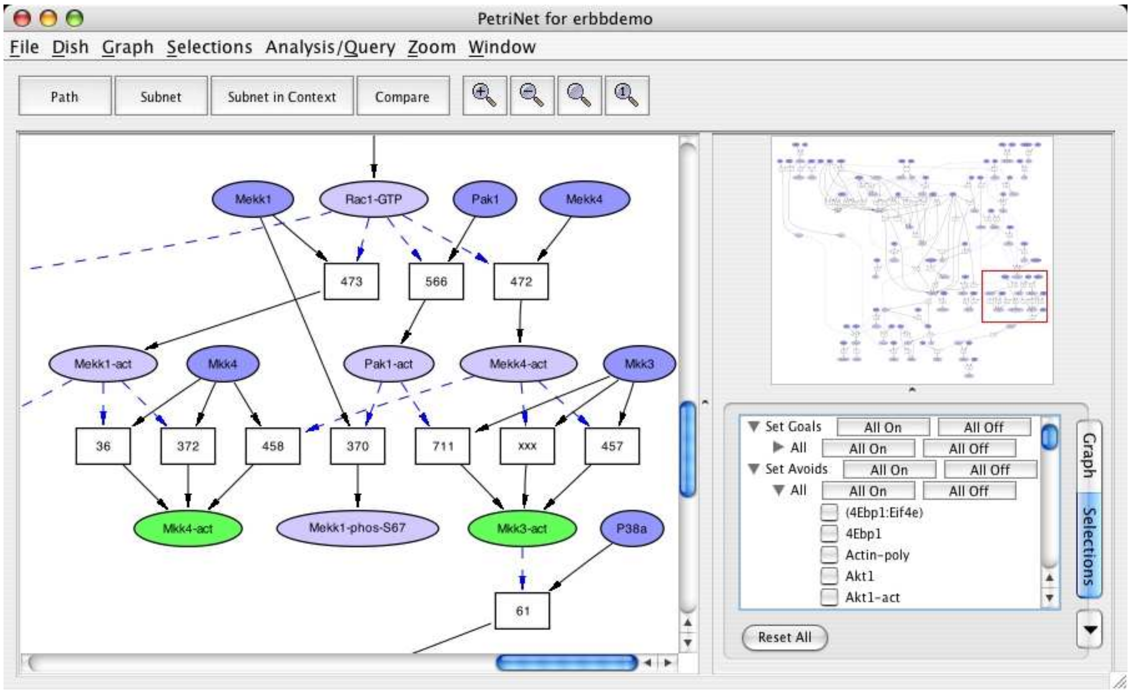


Figure 5: Screen shot of the Pathway Logic Assistant

are zooming capabilities and context information that appears in another tab on the right when users click on nodes and edges of the graph in the bigger view.

Figure 6 depicts the implementation architecture of the Pathway Logic Assistant software. Based on IOP, the aforementioned two actors wrapping `JLambda` and interactive Maude are the main components of PLA. The Maude actor maintains a data base of biological entities and processes written in the Maude language. On the other side, the actor containing the `JLambda` interpreter creates the graphical user interface. For a complex GUI such as PLA, part of the software is implemented in Java, subclassing some of the visualization classes explained in Section 4 to specialize functionality and improve runtime performance. These customized Java classes are bundled into a JAR library that gets loaded at startup time of IOP. Then, we use `JLambda` code to drive the GUI creation and to facilitate the interaction with Maude and its knowledge base through IOP. We take advantage of `JLambda` being interpreted in that smaller software changes, bug fixes, and quickly adding new features for the biologist user can be done without rolling out a recompiled JAR and software installer. Instead, we rather modify `JLambda` source code files and distribute these to the end user who can simply replace the respective files in his or her installation to enjoy the improved software suite.

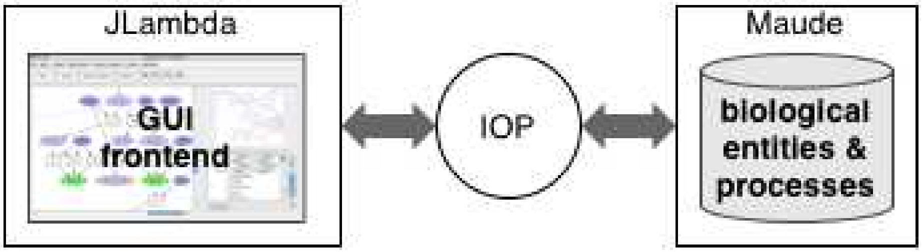


Figure 6: Implementation architecture of the Pathway Logic Assistant

More information about Pathway Logic is available at [30].

6 Conclusion

JLambda has been relatively stable for over a year now, and as mentioned in this paper is a part of the core infrastructure of several ongoing formal modelling research projects. The most recent changes have been adding field updating, and providing interactive debugging capabilities. The class files, as well as a manual are freely available on the web [31]. The sources, protected by the GNU general public licence, are available on request.

References

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

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

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

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

[5] http://www.robert-tolksdorf.de/vmlanguages.html. Programming Languages for the Java Virtual Machine.

[6] Scott G. Miller. SISC: A Complete Scheme Interpreter in Java. Technical report, Indiana University, January 2002.

[7] Per Bothner. Kawa – Compiling Dynamic Languages to the Java VM. In *Proceedings of the Usenix Annual Technical Conference*, June 1998.

[8] K. Anderson, T. Hickey, and P. Norvig. SILK: A Playful Blend of Scheme and Java. In *Proceedings of the Workshop on Scheme and Functional Programming*, pages 13–22, September 2000.

[9] Michael Travers. What is interactive scripting? *Dr Dobb's Journal*, 25:103–110, 2000.

[10] http://xenia.media.mit.edu/~mt/skij/index.html. Skij Homepage, 2004.

[11] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

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

[13] 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.

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

[15] Michael Travers. Scripting and dynamic interaction in Java. Online at http://xenia.media.mit.edu/~mt/skij/dynjava/dynjava.html.

[16] http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4071957. The Java Reflection API Bug.

[17] G. Plotkin. Call by Name, Call by Value, and the Lambda Calculus. *Theoretical Computer Science*, 1, 1974.

[18] M. Felleisen. *The Calculi of Lambda-v-cs Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Indiana University, 1987.

[19] Emden Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with *dot*. www.research.att.com/sw/tools/graphviz/dotguide.pdf.

[20] F. Javier Thayer Fabrega, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7:191–230, 1999.

[21] 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.

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

[23] N. Muscetolla, P. Pandurang, B. Pell, and B. Williams. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence*, 103((1–2)):5–48, 1998.

[24] D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks. Software Architecture Themes In JPL's Mission Data System. In *IEEE Aerospace Conference, USA*, 2000.

[25] G. Denker and C. L. Talcott. Formal checklists for remote agent dependability. In *Fifth International Workshop on Rewriting Logic and Its Applications (WRLA'2004)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2004.

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

[27] 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.

[28] 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.

[29] Merrill Knapp et al. Pathway logic: Helping biologists understand and organize pathway information. In *Poster abstracts of IEEE Computational Systems Bioinformatics Conference (CSB'05)*, pages 155–156, August 2005.

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

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