Prova: Rule-based Java Scripting for Distributed Web Applications: A Case Study in Bioinformatics

Alex Kozlenkov[2], Rafael Penaloza[1], Vivek Nigam[1], Loic Royer[1], Gihan Dawelbait[1], and Michael Schroeder[1]

[1] Biotec, Dresden University of Technology, Germany, contact: ms@biotec.tu-dresden.de
[2] Dept. of Computing, City University, London, UK

Abstract. Prova is a language for rule-based Java scripting to support information integration and agent programming on the web. Prova integrates Java with derivation and reaction rules supporting message exchange with various protocols. Prova supports transparent access to databases, retrieval of URLs, access to web services, and querying of XML documents. We briefly illustrate Prova and show how to implement a distributed bioinformatics application, which includes access to an ontology stored in a database and to XML data for protein structures. Finally, we compare Prova to other event-condition-action rule systems.

1 Introduction

Prova is a language for rule based Java scripting for information integration, and agent programming [6, 3]. Prova is suitable for use as a rules-based backbone for distributed web applications in biomedical data integration. It has been designed to meet the following goals:

- Combine the benefits of declarative and object-oriented programming;
- Combine the syntaxes of Prolog and Java to appeal to programmers in both worlds;
- Expose logic and agent behaviour as rules;
- Access data sources via wrappers written in Java or command-line shells like Perl;
- Make all Java API from available packages directly accessible from rules;
- Run within the Java runtime environment;
- Enable rapid prototyping of applications;
- Offer a rule-based platform for distributed agent programming with common messaging protocols

These design goals are especially important for integration tasks where location and format transparency are important. The latter means that the language should support the work with databases, RDF, HTML, XML, and flat file formats and computational resources alike. Prova's rule-based approach is particularly important for two applications: derivation rules to reason over ontologies and reaction rules to specify reactive behaviour of possibly distributed agents.

Let us consider examples to illustrate these two types of rules. As a declarative language with derivation rules Prova follows a Prolog-style syntax as the next example shows:

Example 1. (Declarative programming)
Graph traversal is a typical examples for declarative programming. A standard example is the same generation problem, in which all nodes in a tree are returned, which belong to the same generation. Two nodes are in the same generation if they are siblings or if their parents are in the same generation. The corresponding Prova programme is identical to standard Prolog.

Listing 1.1. Prova example

```
parent(anna, gerda).
parent(anna, fritz).
parent(asif, anna).
parent(asif, yanju).
parent(yanju, anja).

sg(X,Y) :- parent(Z,X), parent(Z,Y).
sg(X,Y) :- parent(Z1,X), parent(Z2,Y), sg(Z1,Z2).
```

The query `:- solve(sg(gerda,X)).` will return `X=gerda`, `X=fritz`, and `X=anja`.

Thus, Prova follows classical Prolog closely by declaratively specifying relationships with facts and rules. Now let us consider two examples, where access to Java methods is directly integrated into rules.

Example 2. (Object-oriented programming)
The code below represents a rule whose body consists of three Java method calls: the first to construct a String object, the second to append something to the string, and the third to print the string to the screen.

Listing 1.2. Prova example

```
hello(Name):-
  S = java.lang.String("Hello "),
  S.append(Name),
  java.lang.System.out.println(S).
```

2 Prova and Reactivity

Prova's reaction rules can comprise events, conditions, and actions in any order, as both events and actions are reaised by built-in predicates for receiving and sending messages. Both allow for various protocols such as the agent messaging language Jade, the Java messaging system JMS (`java.sun.com/products/jms/` or even Java events generated by Swing components. Due to the natural integration of Prova with Java, Prova's reaction rules offer a syntactically economic and compact way of specifying agents behaviour while allowing for efficient Java-based extensions to improve performance of critical operations. JMS in general has the advantage of being a guaranteed delivery messaging platform. Intuitively it means that when computer A sends a message to computer B the latter is not required to be operational. Once B goes online the messages will be delivered.

2.1 Main features of Prova's reaction rules

Prova provides three main constructs for enabling agent communication:

- sendMsg predicates, which can be used as actions anywhere in the body of a derivation or reaction rule,
- reaction rules, which have a blocking rcvMsg in the head and which fire upon receipt of a corresponding event, and
- inline reactions, which are encoded by blocking receipt of messages using rcvMsg or rcvMult anywhere in the body of derivation or reaction rules.

Communiction actions with sendMsg. The sendMsg predicate can be embedded into the body of an arbitrary derivation or reaction rule. It can fail only if the parameters are incorrect and the message could not be sent due to various other reasons including the dropped connection (note that in the JMS case, the message may be sent anyway even if the network is down).

The format of the predicate is:

```
sendMsg(Protocol,Agent,Performative,[Predicate|Args]|Context)
```

or

```
sendMsg(Protocol,Agent,Performative,Predicate(Args)|Context)
```

where Protocol can currently be either jade, jms, self, or queue. Jade and JMS use the corresponding communication protocols, while self and queue send the message to the agent itself or to another agent running locally in the same process but in another thread. Agent denotes the target of the message. For the self, jade, and jms methods, Agent is the name of the target agent. For the queue option, Agent is the object representing the message queue of the target agent. For Jade messages, the agent name takes the form agent@machine while for jms messages the agent locations are read from configuration files and are not specified in the Agent parameter. Performative corresponds to the semantic instruction the broad characterisation of the message. A standard nomenclature of performatives is FIPA Agents Communication Language ACL (www.fipa.org).

[Predicate|Args] corresponds to the bracketed form and Predicate(Args) corresponds to functional form of the message content sent in the message envelope. The first form can be useful to match any literal including arity-0 predicates (in which case, query() is the represented as [query]) or arity-1 predicates (in which case, query(arg1) is represented as [query,arg1]). The problem with the functional form is that it is impossible to specify a general pattern accommodating predicates of arbitrary arity while the bracketed version is compatible with any arity. Context includes an arbitrary length list of comma-separated parameters that can be used to identify the message or to distinguish the replies to this particular message from other messages. In particular, it can be useful to include the protocol as part of context for the recipient of the message to be able to reply by using the same protocol.

The following code shows a complete rule that sends a code base (a fragment of Prova code) from an external File to the agent Remote that will then assimilate the rules

being sent. The rules are encapsulated in a serializable Java StringBuffer object and sent with the literal for the built-in predicate consult. The particular version of consult will then read on the Remote machine the Prova statements from a StringBuffer (in contrast to the standard version of consult that reads statements from the specified file provided as an input string).

Listing 1.3. sendMsg Example

```
1
2 % Upload a rule base read from File to the host at address Remote
3
4 upload_mobile_code(Remote,File) :-
5         % Opening a file returns an instance of java.io.BufferedReader in Reader
6         fopen(File,Reader),
7         Writer = java.io.StringWriter(),
8         copy(Reader,Writer),
9         Text = Writer.toString(),
10        % SB will encapsulate the whole content of File
11        SB = StringBuffer(Text),
12        sendMsg(jade,Remote,eval,consult(SB)).
```

Reaction rules with rcvMsg. Reaction rules are derivation rules whose head consists of a rcvMsg predicate, which has the same syntax as the sendMsg predicate:

```
rcvMsg(Protocol,To,Performative,[Predicate|Args]|Context)
```

The agent reacts to the message based on its pattern including the protocol, sender, performative, message content, and context. The following code shows a general purpose but simplified reaction rule for the FIPA queryref performative. The first rule triggers a non-deterministic derivation of the literal [Pred|Args] sent as the message content. Based on the agent's knowledge-base derive will instantiate Pred|Args and send corresponding replies. The second rule sends a special end_of_transmission message to inform the querying agent of the completion of the query. The Protocol parameter available as the first parameter allows the recipient of queryref to know the protocol (jade, jms etc.) that should be used for replies.

Listing 1.4. rcvMsg Example

```
1 % Reaction rule to general queryref
2 rcvMsg(Protocol,From,queryref,[Pred|Args]|Context) :-
3         derive([Pred|Args]),
4         sendMsg(Protocol,From,reply,[Pred|Args]|Context).
5 rcvMsg(Protocol,From,queryref,[Pred|Args],Protocol) :-
6         sendMsg(Protocol,From,end_of_transmission,[Pred|Args]|Context).
```

Now we will show how to deploy Prova's derivation and reaction rules to implement a distributed web-based bioinformatics application.

3 The GoProtein tool

Biological databases are growing rapidly. Currently there is much effort spent on annotating these databases with terms from controlled, hierarchical vocabularies such as the GeneOntology [5]. It is often useful to be able to retrieve all entries from a database,

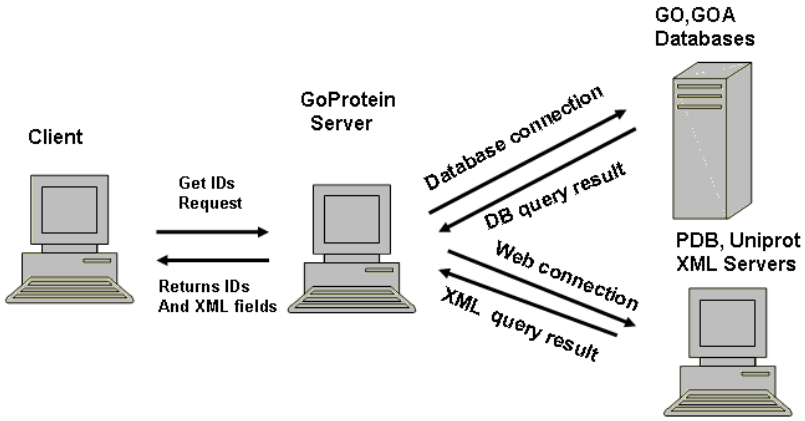

Fig. 1. Sketch of the GoProtein tool workflow: The user interacts locally with a GUI on a client machine. Queries for all proteins annotated with a given term from the ontology are sent to the server. The server can access a database server to obtain protein IDs, which are annotated with the given term. The remote protein database returns an entry for protein as XML file given the protein ID. The protein database is used to display relevant information to the user.

which are annotated with a given term from the ontology. We want to build such a query engine according to the scheme shown in Fig. 1. The application consists of four agents, whose interaction is driven by reaction rules. The agents are a thin client, which contains nothing but a GUI to interact with the user, a server, which handles queries of the client, a database server, which contains the ontology and the protein IDs annotated with the ontology terms, and a protein database which contains detailed information on the protein in XML format. The client's GUI displays the ontology. If the user selects a term from the ontology, an event is fired, which triggers a request being sent to the GoProtein server. The server in turn queries the GeneOntology database server for protein IDs, which have been annotated with the ontology term. If the user selects a specific protein on the GUI, a query is sent to the server, which reacts by retrieving an XML file from the remote protein database and by extracting relevant information from the file and returning it to the client.

For this specific implementation of the GoProtein workflow we want to use the GeneOntology [5] as annotation vocabulary and the Protein Databank PDB [1] as protein database. The Gene ontology (GO) contains over 19.000 terms organised in three sub-ontologies relating to biological processes, molecular functions, and cellular components. GeneOntology is available in XML, OWL, or database dump. Here we use the database dump of the GeneOntology. The protein databank PDB is a database with over 25.000 3D protein structures. Entries contain protein names, species, literature references, and most important the 3D coordinates of all the atoms of the protein. PDB is available as fat file format and XML.

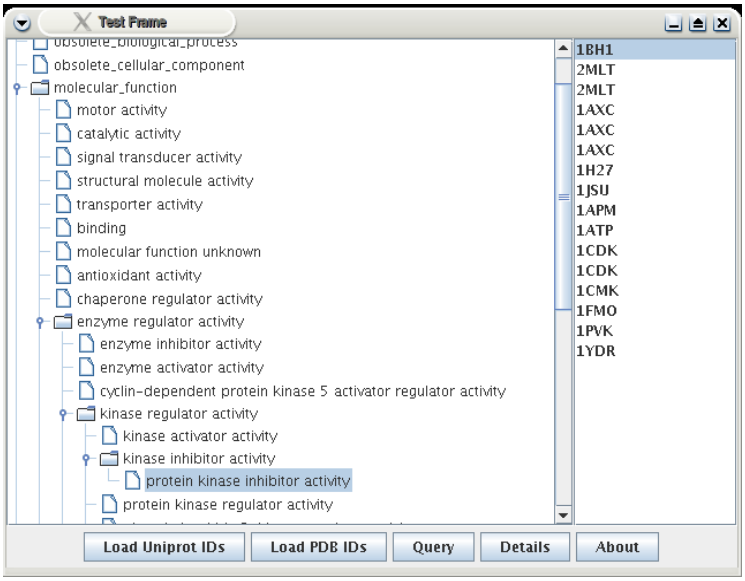

Fig. 2. Screenshot of GoProtein: The left panel shows the ontology including the term ”protein kinase inhibitor” and on the right the PDB entries annotated with the term.

4 Prova code for GoProtein

The client agent comprises the GUI and therefore makes heavy use of Java’s Swing methods. For example, the MutableTreeNode class is used to display the GeneOntology tree.

Listing 1.5. Client agent

```
gui() :-
        println(["==========Window Loading=========="]),
        % create the tree and a placeholder for the IDs
        Node1 = javax.swing.tree.DefaultMutableTreeNode("all"),
        TreeModel = javax.swing.tree.DefaultTreeModel(Node1),
        Tree = javax.swing.JTree(Node1),
        Panel1 = javax.swing.JScrollPane(Tree),
        IdList = javax.swing.JList(),
        Panel2 = javax.swing.JScrollPane(IdList),
    ...
```

For large knowledge bases such as the 19.000 GeneOntology terms it is important to keep data on disc and load it into main memory only as needed. For this purpose, the code snippet below defines the location of the database and uses built-in predicates such as dbopen to open a database connection and sql_select to issue database queries. The concat statements are used to assemble the query string.

Listing 1.6. The Server Agent

```

:- eval(consult("utils.prova")).
```

```
3
4 % Define database location
5 location(database,"GO","jdbc:mysql://myserver","guest","guest").
6
7 % get description of term
8 desc(Term, Desc) :-
9         dbopen("GO",DB),
10        unescape("\'", Quote),
11        concat(["term_definition.term_id = term.id AND term.name =", Quote, ...
              ...Term, Quote],A),
12        concat(["term, term_definition"],From),
13        sql_select(DB,From,['term_definition.term_definition',Desc],[where,A]).
```

The user can also issue a request to extract specified fields from URLs of XML entries for a selected term. The code below shows the ability of Prova to connect to different URLs, process their XML contents and retrieve the requested fields using the built-in prediacte descendantValue.

Listing 1.7. XML Handler

```
1
2 searchPDB("http://pdbbeta.rcsb.org/pdb/displayFile.do?structureId=").
3
4 % get the xml file
5 searchPDB(Query,XML):-
6         print(["Query for ",Query," at PDB"]),
7         searchPDB(BaseURL),
8         concat([BaseURL,Query, "&fileFormat=xml"],URLString),
9         retrieveXML(URLString,XML),
10        println(["done"]).
11
12 % search for "sequence length" values in the xml file of a PDB ID
13 doSearchPDB(Term, Lst):-
14        searchPDB(Term,XML),
15        PDB = "PDBx:",
16        concat([PDB,"length_a"],La),
17        descendantValue(XML,La,A),!,
18        concat([PDB,"length_b"],Lb),
19        descendantValue(XML,Lb,B),!,
20        concat([PDB,"length_c"],Lc),
21        descendantValue(XML,Lc,C),!,
22        Lst = [A,B,C].
23
24 %%%%%%%%%%%%%%%% UTILITIY predicates %%%%%%%%%%%%%%%%%%%%%%
25
26 retrieveXML(URLString,Root):-
27        URL = java.net.URL(URLString),
28        print(["."]),
29        Stream = URL.openStream(),
30        print(["."]),
31        ISR = java.io.InputStreamReader(Stream),
32        XMLResult = XML(ISR),
33        Root = XMLResult.getDocumentElement(),
34        print(["."]).
```

The communication between the client and server agents is performed by using the Prova massaging and reaction rules to specify behaviour of the two agents. The predicate remote in line 1 takes as an argument the specification of the target machine we are communicating with. The reaction rule in line 5, 8, 10 are triggered by an event from the GUI's Swing component, while the one on line 15 is triggered by a message sent by the server. One of the actions triggered by the reaction rule in line 5 is a message sent to the server (last line).

Listing 1.8. The Client Agent

```
remote("ils_assign_server@servername").

% message transfer for the listeners:
% Reaction to button actions
rcvMsg(XID,Protocol,From,swing,[action,Cmd,Source|Extra]) :-
        process_button(Source,Cmd).
% Reaction to incoming swing mouse clicked messages.
rcvMsg(XID,Protocol,From,swing,[mouse,clicked,Src|Extra]) :-
        process_mouse(clicked,Src|Extra).
rcvMsg(XID,Protocol,From,swing,[mouse,entered,Src|Extra]) :-
        process_mouse(entered,Src|Extra).

% message transfer with the server
% actions after receiving the results of a query
rcvMsg(XID,Protocol,From,reply_qry,[IDs]|Context) :-
        mainlist(List),
        buildList(List, IDs).

% process executed when the "Load Uniprot IDs" button is clicked
% it finds the selected node, finds all its associated Uniprot IDs, and loads ...
     ...them in the List
process_button(Button, "Load Uniprot IDs") :-
        Tree = Button.getTree(),
        Path = Tree.getSelectionPath(),
        Node = Path.getLastPathComponent(),
        Term = Node.toString(),
        List = Button.getList(),
        buildList(List, ["contacting server...","please wait"]),
        % ask for the list of associated uniprod IDs
        remote(Remote),
        sendMsg(XID,jade,Remote,uniprot,[Term],"context").
```

The complete code is available upon request and can be demonstrated at the workshop.

5 Comparison and Conclusion

The World Wide Web is a rich heterogenous media following a pattern of growth that is uncentralized, directed by trends, and resistant to initiatives to enforce strong conformance to standards. A language for reactivity on the Web should be simple, offer "'out of the box"' ability to handle most current de facto standards and offer specification robustness through clear declarative semantics. The many recent efforts that have been initiated to bring proper semantics to the Web - The Semantic Web - must also be kept in mind, as they delineate what the Web could eventually become. One can thus enumerate some "'must have"' features for a Reactive Web Language:

- Ability to read and write XML, RDF, OWL, RSS and their variants;
- Possibility to interface to systems written in Java and/or embed java code;
- Connectivity to public/private databases, through different media (direct, web or web-services);
- Simple access to URL-based resources: Web page, Xml file, RSS feed;
- Reactivity through the listening, processing and sending of events and actions;
- Declarative semantics and Event-Condition-Action paradigm.

In the following, we briefly compare Prova with other languages to address the problem of reactivity on the web: JESS, a java based rule engine, XChange based on the Xcerpt web query language, and ruleCore, an XML-based active rule engine.

5.1 JESS

Jess is a forward-chaining rule engine based on the Rete algorithm for the Java platform [4]. Jess supports the development of rule-based systems which can be tightly coupled to code written in the Java language. The syntax of the Jess language is Lisp-based. Java functions can be called from Jess, and Jess can be extended by writing Java code. Jess rules can be embedded in Java applications. Jess inherits from Java all the XML libraries to read, process and write XML data. However, it does not provide rule-based wrappers that provide these facilities in a transparent manner. The same holds for connectivity to databases: it is possible through Java libraries but not truly integrated in the system. This is one of the main differences between Prova and Jess, Prova has specialzed predicates that allow easy and transparent access to databases, XML data, messaging exchange between agents and even to Swing components. The fact that Jess is essentially a rule engine, provides a very natural setup to write Event-Condition-Action rules in the context of event propagation in the Web. Thus Business rules can be stated in a declarative and transparent manner.

5.2 XChange

XChange is a declarative language built upon the declarative web query language Xcerpt[2]. It provides Web-specific capabilities such as propagation of changes on the Web and event-based communications between Web sites. It is a work in progress and as such does not yet have a production-ready implementation. Among the interesting characteristics of XChange is its use of explicit temporal constructs to describe sequences of events, their overlapping and composition. Transactions are also explicit in the language, with the goal of bringing ACID properties to the Reactive Web. Reactivity is achieved by having Event-Condition-Action rules at the core of the language. The main difference between Prova and Xchange is that Prova is a full featured programming language built-upon the robustness and richness of Java, whereas Xchange is geared towards XML and HTML contents.

5.3 ruleCore

Of proved industrial strength, the ruleCore Engine`www.rulecore.com/` is a robust implementation of an active rule engine server. The ruleCore Engine implements Event-Condition-Action rules that are organised in situation trees. The goal of ruleCore is to detect situations that arise as the temporal and logical composition of events. The rule engine itself does not rely on a generic programming language as in the case of Prova and Jess, but instead on the definition of situations as event detector trees. Connectivity to other media and systems is achieved through the use of event and action wrappers, most of which are provided "'out of the box"' for databases and standard industrial messenging frameworks like XML-RPC, Web Services, TIBCO Rendezvous, plain sockets or IBM WebSphere MQ. The main difference between Prova and the ruleCore engine is, as in the case of Xchange, that Prova is a generic rule language extending Java, whereas ruleCore is a language-independent rule engine.

Prova is the choice of a Java programmer with Prolog experience who aims to develop a system which needs a possibly thin layer of rules for reasoning with backward chaining and for defining business rules and workflows with agent communication. Prova is available at www.prova.ws.

References

1. H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne. The protein data bank. *Nucleic Acids Res*, 28(1):235–242, 2000.
2. Franois Bry and Paula-Lavinia Patranjan. Reactivity on the web: Paradigms and applications of the language xchange. In *Proceedings of 20th Annual ACM Symposium on Applied Computing (SAC'2005)*. ACM, March 2005.
3. Jens Dietrich, Alexander Kozlenkov, Michael Schroeder, and Gerd Wagner. Rule-based agents for the semantic web. *Journal on Electronic Commerce Research Applications*, 2(4):323–38, 2003.
4. Ernest Friedman-Hill. *Jess in Action Java Rule-based Systems*. Manning, 2003.
5. GeneOntologyConsortium. The Gene Ontology (GO) database and informatics resource. *Nucleic Acids Res.*, 1(32):D258–61, 2004.
6. Alexander Kozlenkov and Michael Schroeder. PROVA: Rule-based Java-scripting for a bioinformatics semantic web. In E. Rahm, editor, *International Workshop on Data Integration in the Life Sciences DILS*, Leipzig, Germany, 2004. Springer.