Prova Rule Language Version 3.0
User's Guide

Prova is an economic and efficient open source rule language for creating distributed collaborating agents. Running in Java runtime, it combines Prolog style logic inference with extensions for Java scripting, concurrent and scalable reactive messaging, workflows and event processing suitable for Enterprise Service Bus deployment.

Alex Kozlenkov
May 2010

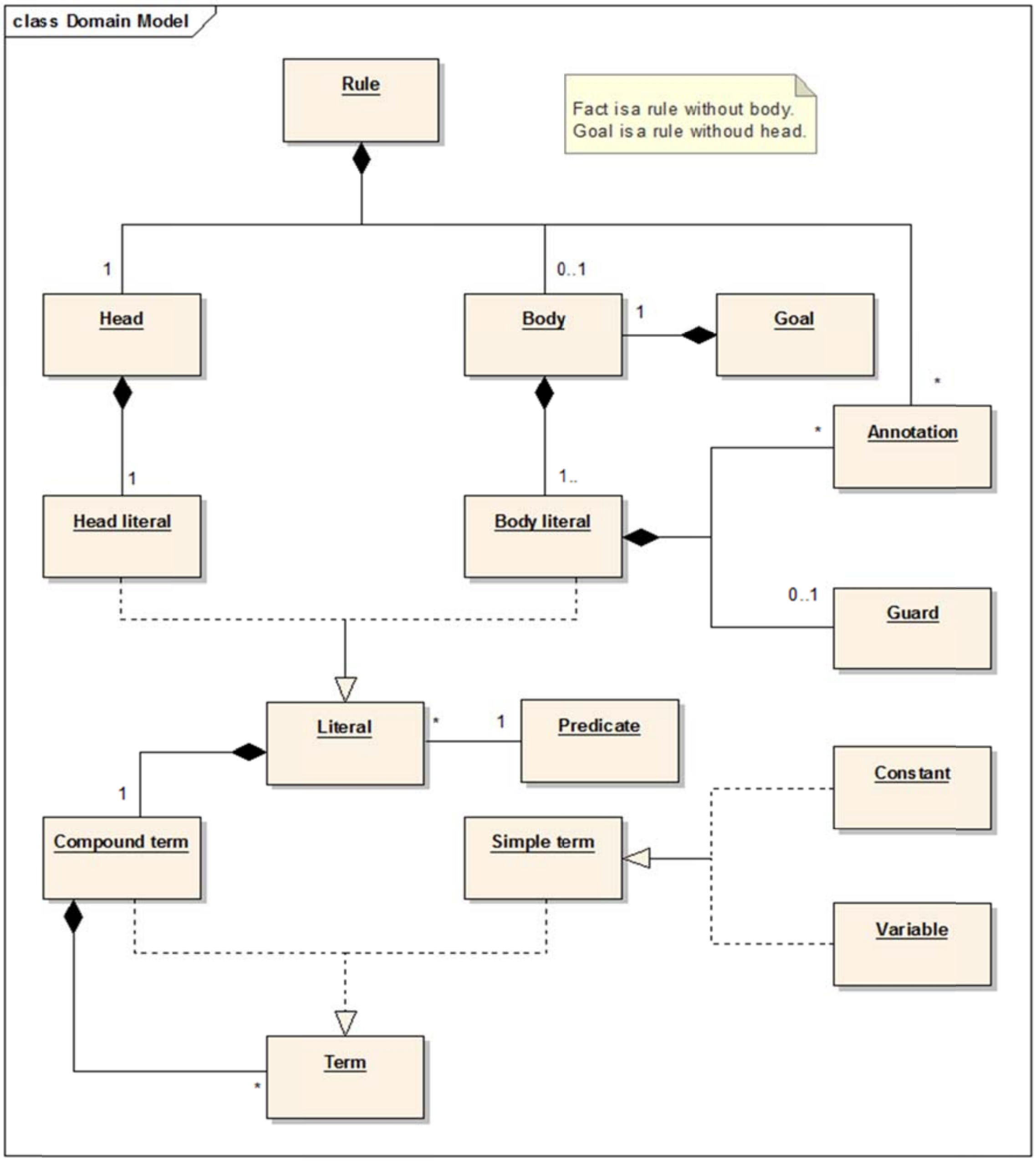
1. General language features

1.1. Basic language elements

Although the roots of Prova lie in logic programming (LP), the similarity can be deceptive, so the Reader will need to keep an open mind and accept that syntactic structures familiar from LP, are given additional semantics and can do much more than in a typical Prolog-like language.

The syntactic simplicity of the language is entirely intentional and motivated by the idea that simple syntax is easier to comprehend and most importantly, easier to scale up and back down, with new syntactic refinements easily available via linear code changes as opposed to complicated refactoring. The ideal here is that of a developer thinking about a new functionality and adding it incrementally, with minimal changes to the existing code. The language is thus far closer to scripting languages that foster quick experimentation and yet, in the best examples, like Ruby, yield high-quality executable systems.

The following diagram captures the main language elements.



At the top level, Prova offers three basic syntactic structures for putting together a rulebase:

rules,
facts (rules with no body),
goals (rules with no head).
These are built at the elemental level from
atomic terms,
compound terms (lists).
Let us talk about them in a bottom-up fashion.

1.1.1. Simple (atomic) terms

Prova includes two types of simple terms: constants and variables. Internally, they are Java interfaces *ProvaConstant* and *ProvaVariable*, respectively.

1.1.2. Variables

Prova variables are essentially hollow pointers that are yet to be given some value assignment. Once the value is assigned, the variable becomes a constant term, that is a data wrapper around a Java object. Once a constant is assigned, in a typical logic and functional programming way, it cannot be assigned some other data. However, if a failure (inability to prove a new sub-goal) is detected during the goal evaluation, backtracking may result in alterantive branching, and therefore, alternative assignments to variables. Assignment of values to variables ultimately is the main *point* of running a Prova rulebase, i.e., providing answers to questions (goals) that are being asked.

As opposed to typical Prolog systems, variables can be typed (see the discussion in Calling Java from Prova rulebases). Previous versions of Prova also allowed variables to be typed using OWL ontologies. This feature is not available in Prova 3.0 but will return in the next version.

Syntactically, as common in Prolog, variables are represented by tokens starting with a capital letter, in the case of typed variables, prefixed by fully qualified package and class name of the corresponding Java class. It is also possible to use anonymous variables that begin with the underscore character. Some examples of Prova variables follow.

```
A
Payload
Integer.I
com.betfair.prova.domain.Market.M
_
```

1.1.3. Constants

Constants may include numeric data using the Java syntax, strings in single or double quotes, as well as fully qualified static or instance fields in Java objects. Single word strings that begin with lower-case letter are for all purposes the same as such strings in quotes.

```
12
-300L
-3.0e-9
test
'test'
"test"
```

It is important to understand that during the execution of a Prova rulebase, variables get assigned values essentially *becoming* constants. In this case, however, these variables are syntactically represented as variables, yet for all purposes, behave like constants. For example, going slightly beyond the scope of this sub-section, this assignment sets the variable to a constant, and subsequent re-assignment of the same variable fails.

```
I=2,
I='cannot assign'
```

Special global constants have names starting with '$' as below.

```
$Count.addAndGet(Delta)
```

1.1.4. Compound terms (lists)

Compound terms are in Prova represented as generic Prova lists. This lends itself well to agent programming as any information can be easily distributed with agents sending each other Prova lists. Critically, and uniquely different from typical LP languages, Prova keeps lists in arrays instead of recursive lists.

Lists are represented in a standard Prolog way as

[Head|Rest]

Here *Head* is one or more (comma-separated) elements that are the list's starting (prefix) elements. The *Rest* is the list tail, that most often is just a variable that will match against the remainder of the list. The vertical bar is then the same as the *cons* operator in functional languages. Lists are an invaluable tool for pattern matching, so that we can say, we want those elements to be there in the list and *unification*, which is the core operation in Prova, will match the lists and assign values to free variables as appropriate.

1.1.5. Facts

A Prova fact is essentially a relation tuple (a record) with its elements being any term, be it constant, variable, or a (possibly recursive) list. This is different from standard Prolog, where facts normally cannot have variables. Moreover, facts can have embedded Java objects, or even global constants. Facts have the following format.

predicate(arg1,...,argn).

It is often the case that there are more than one fact for a given predicate symbol.

alert_destination(no_bids,store2).
alert_destination(no_bids,store_manager2).

1.1.6. Rules

Prova rules *typically* are Horn rules that are first-order disjunctions of literals with exactly one positive literal. In logics, such Horn rules are known as *definite clauses*. The word *typically* highlights the fact that in Prova, *global reaction rules* look exactly like Prova rules (with head literal for predicate symbol *rcvMsg*) but their semantics are more aligned with reactive rules rather than derivation rules.

Rules are written in the standard Prolog like way. Where ':-' is pronounced as IF.

```
head_literal(args_h) :-
   body_literal1(args_1),
   ...,
   body_literaln(args_n).
```

1.2. Running Prova scripts from command line

The new Prova main() is hosted by the ***ProvaCommunicatorImpl*** class. Currently, it requires the following command-line arguments:

1. agent name,
2. password (unused for now),
3. starting rulebase (with possibly included goals that are immediately run synchronously).

Optionally, the timeout for the agent could be specified after that with ***-t <timeout in seconds>***. Specifying the option ***-t -1*** allows the agent to run indefinitely, accepting goals as messages and responding by other actions. If the timeout parameter is not provided, the engine executes all goals in the starting rulebase and then if no internal messages were sent (as is the case with a typical rulebase that has no message sending or receiving), it exits to the command shell. Otherwise, it looks for one second long period of inactivity when no messages are detected and **then** exits to the command shell.

The binary Prova distribution allows running the new Prova interpreter from command line using

```
prova3.bat [-t <timeout>] <rulebase> arg0 arg1 ... argN
```

The list of command-line arguments is passed to the starting rulebase as global constants: $0,$1..,$N.

Here is an example of the arguments passed to the runner:

agent password test001_args.prova anticoagulant

The following rulebase ***test001_args.prova*** then prints one solution ***Parent=molecular_function***.

```
is_a("anticoagulant","molecular_function").

% Rules (how to derive new knowledge)
parent(X,Y) :-   % X is parent of Y IF Y is a (kind of) X
   is_a(Y,X).
parent(X,Y) :-   % X is parent of Y IF X has a Y
   has_a(X,Y).

% Goal (what to do/what to derive)
% Who is Parent of "anticoagulant"?
:- solve(parent(Parent,$0)).
```

1.3. Running Prova from Java

Prova is written in Java but clearly, there needs to be a way for a generic Java program to somehow 'initialise' a Prova agent and interact with it. The Prova agent that serves this purpose is *ProvaCommunicatorImpl*, an implementation of the interface *ProvaCommunicator*. In the test directory for the new Prova, *test.ws.prova.test2*, not all tests are created equal. Some work on primitive constructs, explicitly creating variables or rules from scratch. We do not want to do that here so let's create a *ProvaCommunicator* instance and give it a spin. When you create a *ProvaCommunicator*, there is one main question you need to answer, what rulebase the agent will initially run? Let's look at the test *capture_enum()* in *test.ws.prova.test2.ProvaBuiltins1.java* reproduced below:

```
static final String kAgent = "prova";

static final String kPort = null;

@Test
public void capture_enum() {
   final String rulebase = "rules/reloaded/test017.prova";
   final int[] numSolutions = new int[] {2};
        ProvaCommunicator prova = new
ProvaCommunicatorImpl(kAgent,kPort,rulebase,ProvaCommunicatorImpl.SYNC);
   List<ProvaSolution[]> solutions = prova.getInitializationSolutions();
        org.junit.Assert.assertEquals(solutions.size(),1);
   org.junit.Assert.assertEquals(solutions.get(0).length,numSolutions[0]);
   org.junit.Assert.assertTrue(solutions.get(0)[0].getNv("Groups") instanceof ProvaList);
}
```

This is the simplest way to call Prova from Java. We create a *ProvaCommunicatorImpl*, passing it (apart from the unique agent name and port, discussed elsewhere), the rulebase to be consulted and indicating that the agent will be run in a synchronous mode. At the time of writing, this last parameter is in fact, ignored. If the initial rulebase has any goals, they are run right in the ProvaCommunicatorImpl constructor, making their results available when the client calls *ProvaCommunicator.getInititalizationSolutions()*. This method returns a *List<ProvaSolution[]>*, which is a list of solution arrays, one each for each goal inside the initialisation rulebase. So in the current example, we have one goal and two solutions for that goal. Each *ProvaSolution* contains name/value pairs corresponding to variables' binding.The corresponding rulebase is shown below.

```
:- solve(test017(Groups)).
```

```
test017(Groups) :-
   Text="A doodle is a doddle",
   % non-deterministically enumerate regular expression groups
   capture_enum([Text,"(d?)(dl)"],Groups).
```

1.3.1. Passing Java objects to the initialization rulebase

Java objects can be passed to the Prova engine in a variety of ways. In particular, you can embed Java objects directly into the initialization rulebase, i.e., the rulebase consulted when the ProvaCommunicator is constructed, creating a Prova engine instance. The following code from *ProvaBuiltins1Test.java* demonstrates.

```
@Test
public void read_enum() {
    final String rulebase = "rules/reloaded/read_enum.prova";
    final int[] numSolutions = new int[] {5};

    Map<String,Object> globals = new HashMap<String,Object>();
    globals.put("$File", rulebase);
    ProvaCommunicator prova = new
ProvaCommunicatorImpl(kAgent,kPort,rulebase,ProvaCommunicatorImpl.SYNC,globals);
    List<ProvaSolution[]> solutions = prova.getInitializationSolutions();

    org.junit.Assert.assertEquals(1,solutions.size());
    org.junit.Assert.assertEquals(numSolutions[0],solutions.get(0).length);
}
```

This code passes a map *globals* with named constants (in this example, *$File*) to the consulted rulebase *read_enum.prova* shown below.

```
:- solve(test_read_enum_1(Line)).

test_read_enum_1(Line) :-
   fopen($File,Reader),
   read_enum(Reader,Line).
```

The rulebase opens the file given the supplied filename. Any Java objects can be passed to the rulebase in this way. For example, you can directly embed arbitrary Java objects into Prova facts like this.

```
customer(1234,$c1).
```

Dealing with exceptions

If exceptions occur during the initialization, the engine wraps them in a *RuntimeException* with the appropriate underlying exception reported as its cause. The following test shows a failure due to non-existence of the consulted rulebase.

```
@Test
public void initialization_from_nowhere() {
   final String rulebase = "rules/reloaded/NOSUCHFILE.prova";

   try {
       comm = new ProvaCommunicatorImpl(kAgent,kPort,rulebase,ProvaCommunicatorImpl.SYNC);
   } catch (Exception e) {
       final String localizedMessage = e.getCause().getLocalizedMessage();
       org.junit.Assert.assertEquals(
           "Cannot read from rules/reloaded/NOSUCHFILE.prova",
           localizedMessage);
   }
}
```

If there are parsing errors, the wrapped exception is a *ProvaParsingException*. This exception has two fields that provide further information: *src* and *errors*, i.e, the file origin of the rulebase and a collection of actual parsing errors. This collection maps lines and offsets in format *line:offset* to a textual description of the encountered parsing error as reported by the parser. The following code fragment illustrates.

```
@Test
public void initialization_with_parsing_errors() {
   final String rulebase = "rules/reloaded/parsing_errors.prova";

   try {
       comm = new ProvaCommunicatorImpl(kAgent,kPort,rulebase,ProvaCommunicatorImpl.SYNC);
   } catch (Exception e) {
       org.junit.Assert.assertTrue( e.getCause() instanceof ProvaParsingException );
       org.junit.Assert.assertEquals(
           "rules/reloaded/parsing_errors.prova",
           ((ProvaParsingException) e.getCause()).getSource() );
       org.junit.Assert.assertEquals( 5, ((ProvaParsingException) e.getCause()).errors().size());
   }
}
```

Finally, if an exception occurs inside the rulebase, for example, caused by a failed method invocation, the exception is also wrapped in a *RuntimeException*. The following rulebase *processing_errors.prova* throws a *NoSuchMethodException*.

```
:- solve(p(X)).

p(X) :-
   A=java.util.HashSet(),
   A.nomethod().
```

The test below shows how to detect this exception.

```
@Test
public void initialization_with_rulebase_errors() {
   final String rulebase = "rules/reloaded/processing_errors.prova";

   try {
       comm = new ProvaCommunicatorImpl(kAgent,kPort,rulebase,ProvaCommunicatorImpl.SYNC);
   } catch (Exception e) {
       final String localizedMessage = e.getCause().getLocalizedMessage();
       org.junit.Assert.assertEquals(
           "No such accessible method: nomethod() on object: java.util.HashSet",
           localizedMessage);
   }
}
```

Modifying the rules in a running engine

When a Prova engine instance is active, i.e, the ***ProvaCommunicator*** object is successfully constructed, the latter can be used for adding and removing additional facts or rules to the running rulebase. If the new consulted code contains goals, these goals will then be executed.

The example below shows the use of the ***ProvaCommunicator.consultSync*** method to add a new fragment to the running rulebase.

```
String inputRules = "dissemination(Person,public_reports) :- clearance(Person,low).";
BufferedReader inRules = new BufferedReader(new StringReader(inputRules));

try {
    comm.consultSync(inRules, "dissemination-rules", new Object[]{});
} catch (Exception e) {
    // TODO: process exceptions
}
```

The method takes three parameters:

1. either a ***String*** or ***BufferedReader*** with a rulebase fragment in Prova;
2. a ***String*** with the logical name given to the rulebase fragment;
3. an array of Java objects to replace any placeholders embedded in the fragment.

The call may raise exceptions that the user is required to process.

Removing or replacing the code

The name of the rulebase fragment can be used later for removing the previously added fragment.

```
// Remove all clauses consulted for this key
comm.unconsultSync("dissemination-rules");
```

The fragment can be then easily updated by consulting it again with a new version in a separate step.

1.3.2. Passing Java objects to the added rulebase fragment

The source code submitted to ***consultSync*** can contain placeholders in the syntax _0,...,_n that the call will replace with the Java objects supplied in the third parameter in the rulebase.

```
String input = "robot(_0).";
BufferedReader in = new BufferedReader(new StringReader(input));

try {
    comm.consultSync(inRules, "robot-rules", new Object[]{new Robot("Dave")});
} catch (Exception e) {
    // TODO: process exceptions
}
```

This is actually more powerful than what you can do from a rulebase consulted from an external file during initialization because to embed a Java object in a fact would have required explicitly constructing a Java object and adding a fact dynamically using ***assert***.

```
Robot=com.independentrobots.Robot(),
assert(robot(Robot))
```

1.3.3. Submitting goals and obtaining solutions

If the fragment rulebase contains clauses for meta-predicates *eval* or *solve*, they are executed as new goals. The solutions resulting from the *solve* goals (*eval* do not produce any solutions) are returned in a *List<ProvaSolutions[]>* in the same way it works for initialization solutions.

```
String input = ":- solve(dissemination(Person,public_reports)).";
BufferedReader in = new BufferedReader(new StringReader(input));

try {
    List<ProvaSolution[]> resultSets = comm.consultSync(in, Integer.toString(key++), new Object[]{});
    for( ProvaSolution[] resultSet : resultSets ) {
        org.junit.Assert.assertEquals(numSolutions[i++],resultSet.length);
    }
} catch (Exception e) {
    // TODO: Process exceptions
}
```

1.3.4. Sending messages to the running rulebase

There are two ways to send a message to the running Prova rulebase from Java. The low-level *ProvaCommunicator.addMsg* method can be used for sending a message constructed from a Prova term explicitly built from the basic building primitives, such as constants or lists. The example below demonstrates.

```
// Send a hundred messages to the consulted Prova rulebase.
// Processing is done concurrently on threads belonging to the async pool.
for( int i=0; i<100; i++ ) {
    ProvaList terms = ProvaListImpl.create( new ProvaObject[] {
         ProvaConstantImpl.create("test"+i),
         ProvaConstantImpl.create("async"),
         ProvaConstantImpl.create(0),
         ProvaConstantImpl.create("inform"),
         ProvaListImpl.create(new ProvaObject[] {
             ProvaConstantImpl.create("a"),
             ProvaConstantImpl.create(i)
         })
    });
    comm.addMsg(terms);
}
```

The message needs to be compliant with the standard Prova message format described in Sending messages.

Another method *ProvaCommunicator.sendMsg* offers a more direct, but slower, way to send messages:

```
sendMsg(String xid, String protocol, Object receiver, String perf, String payload, Object[] objs)
```

This signature directly corresponds to the standard message format, see Sending messages.

1.3.5. ProvaService—a simpler way to embed Prova agents

This usage idiom is based on a new feature, a *ProvaService* interface and default implementation *ProvaServiceImpl*, designed for both plain Java runtime and OSGi containers.

The host Java code creates and initializes a *ProvaService*, which can hold one or more Prova agents each running in their own Prova engine. In fact, Prova supports two ways of modularization in an OSGi environment: mapping one agent to a separate bundle or running multiple agents in one bundle. In a plain standalone non-OSGi Java process, the latter is the only option, and this Section shows how simple it is to embed multiple agents and have them communicate via messaging between themselves and with the embedding Java code--also considered to be an "embedding agent".

We reproduce below the source code for a standalone runner class *ProvaSimpleService.java* that demonstrates how to use *ProvaService* to create two cooperating agents. An alternative way to achieve that would have been to use a Mule ESB implementation although the example presented below has a net advantage in terms of simplicity and lack of massive numbers of dependencies required by Mule.

```
package ws.prova.examples.runner;

import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.atomic.AtomicInteger;

import ws.prova.service.EPService;
import ws.prova.service.ProvaService;
import ws.prova.service.impl.ProvaServiceImpl;

/**
 * Demonstrate how ProvaService can host multiple Prova agents that communicate with each other
 * using the "osgi" protocol.
 * Note that this runs in plain Java--no OSGi container is actually required.
 *
 * The runner implements EPService for Prova engines to be able to send messages to the runner,
 * not just between themselves.
 *
 * @author Alex Kozlenkov
 */
public class ProvaSimpleService implements EPService {
    static final String kAgent = "prova";
    static final String kPort = null;
    final String sender_rulebase = "rules/service/message_passing/sender.prova";
    final String receiver_rulebase = "rules/service/message_passing/receiver.prova";
    private ProvaService service;

    public ProvaSimpleService() {
        service = new ProvaServiceImpl();
        service.init();
    }

    private void run() {
        // Create two Prova agents in their own engines
        String sender = service.instance("sender", "");
        String receiver = service.instance("receiver", "");

        // Consult one rulebase into each
        service.consult(receiver, receiver_rulebase, "receiver1");
        // Make sure the consumer has access to a global object for counting
        AtomicInteger count = new AtomicInteger();
        service.setGlobalConstant(receiver, "$Count", count);
        service.consult(sender, sender_rulebase, "sender1");

        // Send a message directly from this runner
        Map<String, Integer> payload = new HashMap<String, Integer>();
        payload.put("a", 2);
        // This runner names itself as "runner" agent in this call (see the third argument).
        // This EPService implementation is passed in the last argument.
        // The service will then register an EPService callback so that agents will be able to
        //   send messages to this runner setting the destination to "runner".
        service.send("xid", "receiver", "runner", "inform", payload, this);

        try {
            synchronized(this) {
                wait(2000);
                // Verify that the count had been incremented
                assert(count.get()==2);
            }
        } catch (Exception e) {
        }
        System.out.println("Confirmed that the messages have been received");

        // Bring both engines down: if we had not done that, both agents would have continued
indefinitely
        service.destroy();
    }

    /**
     * This is a callback that is called when a Prova engine (see receiver.prova) sends a message to this
runner
     */
    @Override
    public void send(String xid, String dest, String agent, String verb,
            Object payload, EPService callback) {
        System.out.println("Received "+verb+" from "+agent+" :"+payload);
    }

    public static void main( String[] args ) {
        new ProvaSimpleService().run();
    }
}
```

This runner code uses two rulebases below. This is a "receiver" agent in rules/service/receiver.prova.

```
:- eval(receiver()).

receiver() :-
    rcvMult(XID,Protocol,From,inform,{a->I}),
    println(["Received ",{a->I}]),
    $Count.incrementAndGet(),

    % This only makes sense if run from ProvaSimpleService.java
    sendMsg(XID,osgi,runner,inform,{a->I}).
```

This is a "sender" agent in rules/service/sender.prova.

```
:- eval(sender()).

sender() :-
    sendMsg(XID,osgi,receiver,inform,{a->1}).
```

When the runner code is initialized in the *ProvaSimpleService* constructor, it creates and initializes a *ProvaService* instance. It then creates two Prova agents using the *ProvaService.instance()* method and then uses the method *consult* to add the Prova rulebases to those agents. When the rulebases are consulted, they run their *eval* goals with the sender sending and the receiver receiving a message. For the agents to be "visible" to each other, they must communicate via the "osgi" protocol (the second argument in *sendMsg* and *rcvMsg*).

Using a global object *count* is only needed in this slightly contrived example: we use it for incrementing the count of expected messages (see *$Count.incrementAndGet()* in *receiver.prova*) and checking this count from the Java code.

The example also demonstrates that we can both send messages **to** the selected agents and receive messages back from the Java code. For this to work, the key is to pass a chosen name of the containing "agent" to the _ProvaService.send()_ method:

service.send("xid", "receiver", "runner", "inform", payload, this);

ProvaService registers the name "runner" against the callback object passed as the last parameter. Note that the *ProvaSimpleService* implements the *EPService* interface, allowing it the Prova agents to call back on the *EPService.send()* method when the need to send messages back to the containing agent. Observe the *sendMsg* with destination "runner" in *receiver.prova*.

sendMsg(XID,osgi,runner,inform,{a->I}).

Also, observe the use of Java *Map* for passing message payloads between all parties, including the embedding agent. This improves performance and yet allows for considerable flexibility (see Prova maps and messaging using slotted terms).

This is a typical output from running this program.

```
Prova Service 7faee1de-a6e1-421d-ac2e-97709ba47196 created
==========Service messaging test: receiver==========
==========Service messaging test: sender==========
Received {a=1}
Received {a=2}
Received inform from receiver :{a=1}
Received inform from receiver :{a=2}
Confirmed that the messages have been received
Prova Service 7faee1de-a6e1-421d-ac2e-97709ba47196 destroyed
```

To summarize, we are able to exchange messages between all parties: the Java code representing an embedding agent and the embedded Prova agents. The messages can flow in any direction between all parties with only logical agent names required as destination.

1.3.6. Running Prova inside an OSGi container

Prova 3.0 is packaged as an OSGi bundle that can expose a Prova service and necessary domain classes accessible from other bundles in an OSGi container. We use *META-INF/MANIFEST.MF* reproduced below for declaring the bundle dependencies and declaring packages and services useful for other bundles.

```
Manifest-Version: 1.0
Bundle-Version: 3.0.0
Bundle-Name: Prova compact
Bundle-ManifestVersion: 2
Bundle-Description: Prova rule engine compact
Bundle-SymbolicName: ws.prova.compact
Bundle-RequiredExecutionEnvironment: J2SE-1.5
Import-Package: org.osgi.framework;version="1.3.0",
 org.apache.commons.collections;version="3.2.1",
 org.apache.commons.beanutils;version="1.8.0",
 org.apache.log4j;version="1.2.15"
Require-Bundle: com.springsource.org.antlr.runtime;bundle-version="3.1.3"
Export-Package: ws.prova.service;version="3.0.0",
 ws.prova.api2;version="3.0.0",
 ws.prova.exchange;version="3.0.0",
 ws.prova.exchange.impl;version="3.0.0"
Bundle-ClassPath: .,
 target/classes/,
 rules
Bundle-ActivationPolicy: lazy
```

As we can see, the number of dependencies that Prova relies on is very small. We also use Spring DM context configurations in *META-INF/spring* to make *ProvaService* available to other bundles as a service. When an OSGi target platform starts, it exposes the *ProvaService* as an OSGi service without us having to have any dependency or coupling in the project itself on OSGi or Spring.

You can download the platform from: http://www.prova.ws/downloads/platform.zip. Unzip it to a directory and from Eclipse go to **Window/Preferences/Plug-in development/Target platform** and create and choose a new target platform using the directory you expanded the platform bundles into.

Check out the Prova Eclpse project from https://mandarax.svn.sourceforge.net/svnroot/mandarax/prova3/prova-compact/trunk and run **mvn clean install -DskipTests** from command line. Refresh the project in Eclipse.

You can run the Prova bundle in an OSGi container by first creating a new run configuration by going to **Run/Run configurations.../OSGi Framework** including the Prova bundle from the Prova project in addition to all the bundles included in the imported platform. Running this configuration will output something like this to the console:

```
osgi> 0    [Start Level Event Dispatcher] INFO
org.springframework.osgi.extender.internal.activator.ContextLoaderListener  - Starting
[org.springframework.osgi.extender] bundle v.[1.2.1]
90   [Start Level Event Dispatcher] INFO
org.springframework.osgi.extender.internal.support.ExtenderConfiguration  - No custom extender
configuration detected; using defaults...
99   [Start Level Event Dispatcher] INFO  org.springframework.scheduling.timer.TimerTaskExecutor  -
Initializing Timer
185  [Start Level Event Dispatcher] INFO
org.springframework.osgi.extender.support.DefaultOsgiApplicationContextCreator  - Discovered
configurations {osgibundle:/META-INF/spring/*.xml} in bundle [Prova compact (ws.prova.compact)]
```

```
357  [SpringOsgiExtenderThread-1] INFO
org.springframework.osgi.context.support.OsgiBundleXmlApplicationContext  - Refreshing
OsgiBundleXmlApplicationContext(bundle=ws.prova.compact, config=osgibundle:/META-INF/spring/*.xml):
startup date [Thu Apr 29 13:54:59 BST 2010]; root of context hierarchy
358  [SpringOsgiExtenderThread-1] INFO
org.springframework.osgi.context.support.OsgiBundleXmlApplicationContext  - Application Context service
already unpublished
426  [SpringOsgiExtenderThread-1] INFO
org.springframework.beans.factory.xml.XmlBeanDefinitionReader  - Loading XML bean definitions from URL
[bundleentry://59.fwk1384828782/META-INF/spring/module-context.xml]
755  [SpringOsgiExtenderThread-1] INFO
org.springframework.beans.factory.xml.XmlBeanDefinitionReader  - Loading XML bean definitions from URL
[bundleentry://59.fwk1384828782/META-INF/spring/module-osgi-context.xml]
907  [SpringOsgiExtenderThread-1] INFO
org.springframework.osgi.extender.internal.dependencies.startup.DependencyWaiterApplicationContextExe
cutor  - No outstanding OSGi service dependencies, completing initialization for
OsgiBundleXmlApplicationContext(bundle=ws.prova.compact, config=osgibundle:/META-INF/spring/*.xml)
911  [SpringOsgiExtenderThread-2] INFO
org.springframework.beans.factory.support.DefaultListableBeanFactory  - Pre-instantiating singletons in
org.springframework.beans.factory.support.DefaultListableBeanFactory@57c8b24d: defining beans
[provaService,org.springframework.osgi.service.exporter.support.OsgiServiceFactoryBean#0]; root of
factory hierarchy
Prova Service 090128f4-d886-400a-a631-85088bdd08e0 created
996  [SpringOsgiExtenderThread-2] INFO
org.springframework.osgi.service.exporter.support.OsgiServiceFactoryBean  - Publishing service under
classes [{ws.prova.service.ProvaService}]
999  [SpringOsgiExtenderThread-2] INFO
org.springframework.osgi.context.support.OsgiBundleXmlApplicationContext  - Publishing application
context as OSGi service with properties {org.springframework.context.service.name=ws.prova.compact,
Bundle-SymbolicName=ws.prova.compact, Bundle-Version=3.0.0}
1006 [SpringOsgiExtenderThread-2] INFO
org.springframework.osgi.extender.internal.activator.ContextLoaderListener  - Application context
successfully refreshed (OsgiBundleXmlApplicationContext(bundle=ws.prova.compact,
config=osgibundle:/META-INF/spring/*.xml))
```

This confirms that the Prova bundle started successfully.

1.3.7. Prova agent contract and ESB adaptor

The way to start a Prova engine from Java described in the previous Section does not follow any particular pattern. In particular, there is no way for the Prova rulebase to take an active role and communicate with the embedding Java code, apart from passing a callback object to the rulebase and the latter invoking the callback when the rulebase needs to send data.

Prova 3.0 includes the following simple *ProvaAgent* interface that imposes a contract on the adaptor Java object that embeds a Prova engine instance. In a separate *mule-prova-agents* project, we use a Mule 2.1 Enterprise Service Bus implementation of this contract. This allows Prova agents to be provisioned by the Mule ESB configuration and communicate with each other via a variaty of transports supported by Mule. If another ESB (or equivalent) implementation is required, all that is required is implementing the *ProvaAgent* interface described below.

```
public interface ProvaAgent {
   public void send(String dest, ProvaList terms) throws Exception;
   public String getAgentName();
}
```

We recommend consulting the *Prova3AgentImpl.java* code for an implementation suitable for Mule ESB as it provides a simple example of the required functionality. As described in the Sending messages (in particular, see the discussion on *Protocol*), the agent rulebase sending messages using the "esb" protocol will automatically hit the method *ProvaAgent.send*, which then will publish them into the ESB layer to the indicated destination.

```
@Override
public void send(String dest, ProvaList terms) throws Exception {
   MuleClient client = new MuleClient();
   client.dispatch(dest,terms,null);
}
```

Conversely, all messages received by the agent will indicate the received protocol as "esb" but typically will override that with the "async" protocol to allow for the same conversation-id to be mapped to the same agent thread.

```
public Object onCall(MuleEventContext context) throws Exception {
      MuleMessage inbound = context.getMessage();
      ProvaList terms = null;
      if( inbound.getPayload() instanceof ObjectMessage )  {
        terms = (ProvaList) ((ObjectMessage) inbound.getPayload()).getObject();
      } else {
        terms = (ProvaList) context.getMessage().getPayload();
      }

      // Add the message as a goal to the asynchronous Prova Communicator queue
      if( !"".equals(targetProtocol) ) {
        terms.getFixed()[1] = new ProvaConstantImpl(targetProtocol);
      }
   comm.addMsg(terms);

   // We are done, everything is asynchronous
   context.setStopFurtherProcessing(true);

      return null;
}
```

1.4. Calling Java from Prova rulebases

Prova agents run in Java so it is natural that Prova rulebases can also call home and work with Java data and objects. This integration includes:

- Using Java types and classes for Prova variables;
- Creating Java objects by calling their constructors;
- Passing Prova lists as arguments to Java constructors;
- Invoking static methods, including object creation by using factory methods;
- Accessing public static and instance fields;
- Using special built-ins that take advantage of common Java classes;
- Adding new Prova built-ins in Java.

1.4.1. Using Java types and classes for Prova variables

According to some Prolog experts and implementers, the ansence of variables typing in Prolog is one of the reasons of its relative failure to become completely mainstream. Prova takes a more practical approach by allowing Java types explicitly and clearly annotating variables using a simple syntax as in the following example *ra001.prova*:

```
:- solve(a(3)).
:- solve(a(Number.X)).
:- solve(a(Integer.I)).
:- solve(a(Short.I)).

a(Integer.I).
```

```
a(Double.I).

% ==================
% This prints:
% yes
% X=java.lang.Integer.<7>
% X=java.lang.Double.<10>
% I=java.lang.Integer.<13>
% no
```

Unification that includes variable types follows the following subsumption rules.

1. If the query (goal) is 'asking' for a narrower type, it will not match a wider type in the target literal.
2. A wider query will match a more specialized target literal.

Observe how uninstantiated variables also benefit from this with the second goal *solve(a(Number.X))* matching any applicable rules that are more specialized.

Note that classes in the *java.lang* package do not require to be full qualified. Fully qualified class names can also be used in typed variables. The following code shows how the builtins retract, retractall, and *insert* take into account typed variables.

```
:- solve(ra002_1(X,Y)).
:- solve(ra002_2(X,Y)).
:- solve(ra002_3(X,Y)).

f(ws.prova.examples.IBMPortfolio.N,Integer.M).
f(ws.prova.examples.Portfolio.N,M).
f(ws.prova.examples.IBMPortfolio.N,Integer.M).
f(2,3).

ra002_1(X,Y) :-
   println(["-----"]),
   % The first fact not more general than this is removed by retract below
   retract(f(ws.prova.examples.Portfolio.N,M)),
   f(X,Y).
ra002_2(X,Y) :-
   println(["-----"]),
   % Facts more general than this are not removed by retractall below
   retractall(f(ws.prova.examples.IBMPortfolio.N,Integer.M)),
   f(X,Y).
ra002_3(X,Y) :-
   println(["-----"]),
   insert(f(ws.prova.examples.Portfolio.N,Integer.M)),
   % Does not add anything as it subsumes a fact already in the KB
   insert(f(ws.prova.examples.Portfolio.N,M)),
   f(X,Y).

% ==================
% This prints:
% -----
% X=ws.prova.examples.Portfolio.<30>, Y=<27>
% X=ws.prova.examples.IBMPortfolio.<39>, Y=java.lang.Integer.<40>
% X=2, Y=3
% -----
% X=ws.prova.examples.Portfolio.<80>, Y=<77>
% X=2, Y=3
% -----
% X=2, Y=3
% X=ws.prova.examples.Portfolio.<137>, Y=java.lang.Integer.<138>
% X=ws.prova.examples.Portfolio.<150>, Y=<146>
```

1.4.2. Creating Java objects by calling their constructors

Prova allows Java constructors to be invoked using the familiat Java syntax, only dropping the **new** keyword. Here are a few examples.

```
L=java.util.ArrayList()
DF=java.text.SimpleDateFormat("dd/MM/yyyy kk:mm:ss.mmm")
Calendar=java.util.GregorianCalendar()
```

1.4.3. Passing Prova lists as arguments to constructors

When a Prova list is passed as an argument to a Java constructor, we follow the agreement that the top-level elements in the supplied list are unwrapped if necessary from the Prova constant representation to naked Java objects and the list itself is then shipped to the method (or constructor) as Java list containing these objects. **In the case of ordinary Java method calls, Prova lists are passed to the list unchanged.**

This fragment from test004.prova illustrates.

```
List=java.util.ArrayList([1,"2"])
```

1.4.4. Invoking static methods

Here is a typical example of a static method invocation, in this instance, this is a factory method.

```
Model = com.hp.hpl.jena.rdf.model.ModelFactory.createDefaultModel()
```

1.4.5. Accessing public static and instance fields

Public static fields can be accessed using the familiar Java syntax. This example uses a few public enumerations as well as the standard out stream as a parameter in a method.

```
Child1 = Model.createResource(),
Child1.addProperty(com.hp.hpl.jena.vocabulary.VCARD.Given,GivenName),
Child1.addProperty(com.hp.hpl.jena.vocabulary.VCARD.Family,FamilyName),
JohnSmith.addProperty(com.hp.hpl.jena.vocabulary.VCARD.N,Child1),
Model.write(System.out),
```

1.4.6. Using special built-ins that take advantage of common Java classes

At the moment, there are two built-in predicates that use Java data matching appropriate interfaces.

The element built-in can non-deterministically extract elements from instantiated Java iterators.

```
Iter = Model.listStatements(),
element(Stmt,Iter),
```

The findall built-in accumulates the resultset of a goal in a Java *ArrayList*. For example, the following rule accumulates all the solutions of the goal *[X|Xs]* in the new list *L*, which is then sent back to the caller as a whole in one reply message.

```
% Reaction rule to a general queryref_acc
rcvMsg(XID,Protocol,From,queryref_acc,[ID,[X|Xs]]) :-
   findall([X|Xs],[X|Xs],L),
   sendMsg(XID,Protocol,From,reply,[ID,L]).
```

1.5. Expressions

Prova 3.0 can handle arithmetic expressions on the right-hand side of binary operators. The grammar is shown below.

expr : aterm ((PLUS | MINUS) expr)?;

aterm : (MINUS? variable | number | MINUS? predicate_java_call | OPEN expr CLOSE) ((MULT | DIV | REM) aterm)?;

```
predicate_java_call
   :   static_java_call | instance_java_call
```

```
;
```

It is currently impossible to use expressions in arguments of Java calls. It is also not allowed to use Java constructors in expressions unless in isolation. Here are some examples.

```
expr001(N,M) :-
   N=M+1*2.
expr001(N,M) :-
   N<=-M+1*-2.
expr001(N,M) :-
   N>(-M+1)*-2.

expr002(N,M) :-
   N=java.lang.Math.min(1,M)+1.
expr002(N,M) :-
   L=java.util.ArrayList(),
   L.add(2),
   N=-L.get(0)+L.size().
```

1.6. Global constants

Global constants is a new functionality in Prova. Look at the following example ***globals001.prova***.

```
:- solve(globals(N)).

globals(N) :-
   data($A,N).
globals(N) :-
   $A=b,
   data($A,N).

data(a,1).
data(b,2).
```

Any constant with name starting with the $ sign is a global constant. These constants are part of the _ProvaKnowledgebaseImpl_ instance and can be instantiated prior to parsing a new rulebase or modified (or added) during the execution of the code. This is accomplished by using the syntax ***$Var=<rhs>***.

The code above is run from the new test ***ProvaGlobalsTest.globals001()*** that initially sets ***$A*** to ***'a'*** so it returns two solutions: 1 and 2.

The intended use of global constants ***$NNN*** is for passing command-line arguments to Prova scripts. This functionality will be added once the ***main()*** runner is added to the new Prova code.

1.7. Prova maps for defining slotted terms

Slotted terms is a highly valuable feature of languages like F-Logic, standards (RIF or RuleML), and products (like OO-jDREW). They also make interoperability with forward-chaining rule systems like JBoss Rules far easier.

Prova 3.0 includes an implementation of slotted terms using the W3C RIF arrow expression syntax as in the following example *map2.prova*. Note that Prova also allows to use colon ':' in *key:value* instead of *key->value*. The keys are currently limited to strings.

```
:- solve(own(X)).

buy({buyer->'Bill',seller->'Amazon',item->'Avatar Blu-ray'}).

keep({keeper->'Bill',item->'Avatar Blu-ray'}).

own({owner->Person,item->Object}) :-
```

```
    buy({buyer->Person,seller->Merchant,item->Object}),
    keep({keeper->Person,item->Object}).
```

This shows that Prova maps can be used as sole arguments of predicates, essentially replacing position-based syntax with one based on named arguments. The example prints:

```
X={item=Avatar Blu-ray, owner=Bill}
```

The test *map2()* in *ProvaBuiltins1Test.java* is based on this example.

Slotted terms are especially useful in reactive messaging, see the discussion on slotted terms in reactive messaging.

1.8. Metadata annotations

The old Prova version had an interesting but somewhat patchy approach to run-time processing of the rule metadata. Prova 3.0 is aiming to improve on this. The first question is what to annotate with metadata? It is clear that all Prova clauses (rules and facts) should be allowed to carry metadata. As you'll see below, body literals can also be annotated so that whenever unification occurs, it uses metadata matching if metadata is present on the body literal.

The examples below are taken from a new test ***rules/reloaded/label.prova***. This is the way metadata is added to rules:

```
@label(rule1) r1(X):-q(X).
@label(rule2) r2(X):-q(X).
@label(rule3) r2(X):-q(X).
```

Metadata is a set of annotations each of which is a pair defined as: ***@key(value** [,**value**]*)*. Both keys and values are arbitrary strings defined by their occurrence in an annotation. All rules in a rulebase consulted from a file automatically have a special annotation ***@src(FILENAME)***. Apart from that, all other rule annotations are defined by the user. There could be more than one user-defined annotation associated with a single rule. There is no requirement for annotations to be on the same line as the rule head.

Now there wouldn't be any real use for this if we couldn't somehow take this metadata into account when the rulebase is executed. To achieve that, the body literals in a rule can be optionally annotated as well. Here is an example that returns 3 solutions ***(1,2,3)***:

```
p1(X):-
    @label(rule1)
    r1(X).
q(1).
q(2).
q(3).

:-solve(p1(X2)).
```

The annotations on literals (as opposed to rules) are a set of constraints on the target rules during unification. For each literal annotation, there must be at least one match between a value listed for that annotation and a value listed for the same key in the target rule. This is a more complex example that also has three solutions ***(1,2,3)***:

```
% succeeds since scope is EITHER "rule2" OR "rule1"
p2a(X):-
    @label(rule2,rule1)
    r1(X).
:-solve(p2a(X4)).
```

Imagine now that we do not know what are the target annotation values and would like to discover and possibly trace the annotations for target rules matching a literal. How about trying to use a variable instead of a constant value in a literal annotation? The following returns 6 solutions where ***Label6*** takes values 'rule2' and 'rule3' 3 times each.

```
p4(X,Y):-
    @label(Y)
    r2(X).
```

```
:-solve(p4(X6,Label6)).
```

Observe that if a match is found, execution continues for ***r2(X)*** as though the *@label* annotation was not present.

Now since we have variables in literal annotations, we can also pre-assign values to these variables dynamically. The following code sets the annotation value at run-time to 'rule2' so that only 3 solutions for ***X7*** are returned ***(1,2,3)***.

```
% dynamically set the metadata value expected from matching rules
:-solve(p4(X7,rule2)).
```

Finally, we can also find the rulebase of the target clause in a unification. The final example shows how we can constrain one annotation (*@label*) while enquiring on the value of another (*@src*). In this way, any match can be traced from the source rulebase of the target clause. This returns 3 solutions with ***Src9*** set to the filename of the rulebase in each solution.

```
% get module label
p6(X,Y):-
   @src(Y) @label(rule3)
   r2(X).
:-solve(p6(X9,Src9)).
```

1.9. Guards in Prova

Prova 3.0 implements a new inference extension called literal ***guards***. Let's start with an example. Imagine that during unification, the target rule matches the source literal but we do not want to proceed with further evaluation unless a ***guard*** condition evaluates to true. The guards are specified in a syntax that is similar to the one used in Erlang (using brackets instead of the **when** keyword) but the semantics are more appropriate for a rule language like Prova. See ***rules/reloaded/guard.prova*** and ***ProvaMetadataTest.java***.

```
@author(dev1) r1(X):-q(X).
@author(dev22) r2(X):-q(X).
@author(dev32) r2(X):-s(X).

q(2).
s(-2).

trusted(dev1).
trusted(dev22).
% Author dev22 is trusted but dev32 is not, so one solution is found: X1=2
p1(X):-
   @author(A)
   r2(X) [trusted(A)].
:-solve(p1(X1)).
```

This example uses metadata annotations on rules for predicates ***r1/1*** and ***r2/1*** and on the literal ***r2(X)*** in the body of the rule for ***p1(X)*** (see Metadata annotations). Since variable ***A*** in *@author(A)* is initially free, it gets instantiated from the matching target rule(s). Once ***A*** is instantiated to the target rule's *@author* annotation's value ('dev22', for the first ***r2*** rule), the body of the target rule is dynamically non-destructively modified to include all the literals in the guard ***trusted(A)*** before the body start, after which the processing continues. Since ***trusted(dev22)*** is **true** but ***trusted(dev32)*** is not, only the first rule for predicate ***r2*** is used and so one solution ***X1=2*** is returned by ***solve(p1(X1))***.

Guards can include arbitrary lists of Prova literals including Java calls, arithmetic expressions, relations, and even CUT. The next example shows how CUT can be used in the literal guard.

```
a(X):-qq(X).
a(X):-s(X).

s(-2).
```

```
% This example shows how guards can be used to implement a "dynamic CUT".
% The CUT in the guard is dynamically spliced into the start of the target rule body.
% As qq(X) has no solutions, the CUT prevents further backtracking to the second rule for a(X):-s(X),
%    that would have yielded a solution but does not due to that dynamic CUT.
p2(X):-
   a(X) [!].
:-solve(p2(X2)).
```

If the ***CUT*** was included as part of the ***p2(X)*** rule after ***a(X)*** like in the following version, the ***a(X)*** rule including ***qq(X)*** would have failed but the second rule would have provided one solution: ***-2***.

```
p2(X):-
   a(X),!.
```

Prova guards play even a more important role in message and event processing as they allow the received messages to be examined before they are irrevocably accepted, see the details in Guards in message processing.

1.10. Workflows and business processes

Prova offers a fairly unique combination of processing techniques, including workflows, reactive messaging, event processing and elements of functional programming. The idea is to give the users a great freedom for building distributed agents using a holistic approach that does not force them to a particular methodology.

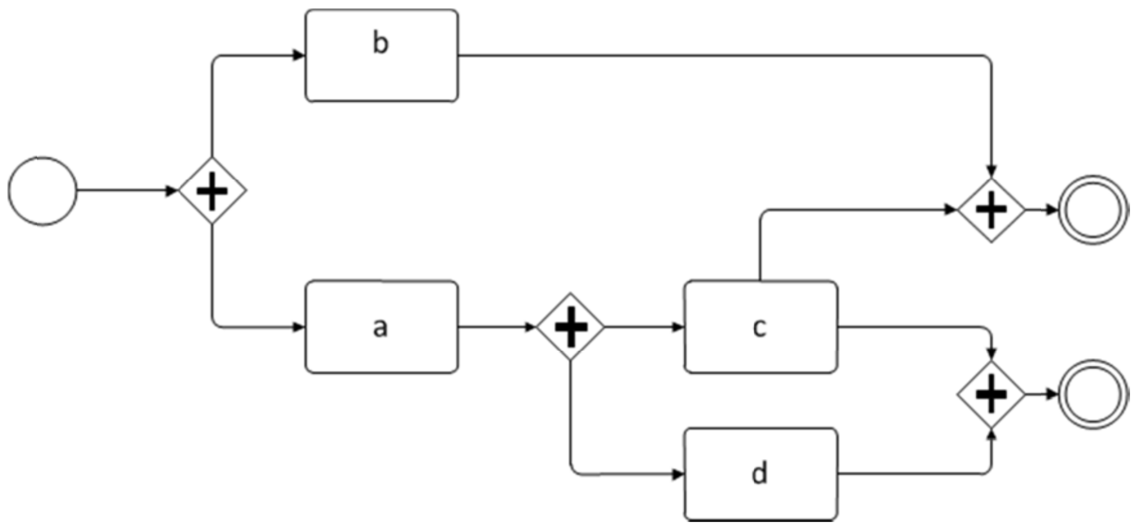
The following syntactic and semantic instruments in Prova 3.0 capture the basics and offer unique advanced features for implementing workflows and business processes.

- inherent non-determinism for defining process divergences;
- Reactive messaging;
- Concurrency support, including partitioned and non-partitioned thread pools;
- built-in predicate spawn for running tasks;
- process join
- predicate join
- reaction groups combining event processing with workflows (see event driven gateways and Event processing using reaction groups);
- support for dynamic event channels;
- guards

This Section discusses the Prova support for workflows, in particular, both simple and sophisticated logic-based *predicate* gateways.

1.10.1. Simple gateways

Prova includes a very simple mechanism (inspired by Join-calculus) for creating diverging branches (parallel gateway) and process join points (inclusive gateways). Consider the following simple workflow presented in the BPMN notation.



The following code from *process_join.prova* uses the built-in predicates *init_join* and *join* to implement this workflow.

```
:- eval(process_join()).

process_join() :-
    println(["==========process_join=========="]),
    % This message signals the start of the conversation. The engine initializes the conversation-id XID.
    sendMsg(XID,self,0,start,[]),
    % The result of the above is that we now have the conversation-id XID initialised.
    % Create a join predicate join_1 with the list of required input patterns in the last argument.
    init_join(XID,join_1,[c(_),b(_)]),
    % Create a join predicate join_1 with the list of required input patterns in the last argument.
    init_join(XID,join_2,[c(_),d(_)]),
    % This will create two parallel processing streams.
    % Note that Prova does not run them in separate threads but they are independent and can
communicate
    %    with other agents asynchronously effectively allowing for parallel processing.
    fork_a_b(XID).

fork_a_b(XID) :-
    % Task a
    sendMsg(XID,self,0,reply,a(1),corr_a),
    rcvMsg(XID,self,Me,reply,a(1),corr_a),
    fork_c_d(XID).
fork_a_b(XID) :-
    % Task b
    sendMsg(XID,self,0,reply,b(1),corr_b),
    rcvMsg(XID,self,Me,reply,b(1),corr_b),
    % Tell the join join_1 that a new pattern is ready
    join(Me,XID,join_1,b(1)).

fork_c_d(XID) :-
    % Task c
    sendMsg(XID,self,0,reply,c(1),corr_c),
    rcvMsg(XID,self,Me,reply,c(1),corr_c),
    % Tell the join join_1 that a new pattern is ready
    join(Me,XID,join_1,c(1)).
fork_c_d(XID) :-
    % Task d
    sendMsg(XID,self,0,reply,d(1),corr_d),
    rcvMsg(XID,self,Me,reply,d(1),corr_d),
    join(Me,XID,join_2,d(2)).

% The following rule will be invoked by join once all the inputs are assembled.
join_1(Me,XID,Inputs) :-
    join(Me,XID,join_2,c(2)),
    println(["Joined ",join_1," for XID=",XID," with inputs: ",Inputs]).

% The following rule will be invoked by join once all the inputs are assembled.
join_2(Me,XID,Inputs) :-
    println(["Joined ",join_2," for XID=",XID," with inputs: ",Inputs]).

% A testing harness catch-all reaction for printing all messages.
rcvMsg(XID,Protocol,From,Performative,[X|Xs],Extra) :-
    println([Performative,[X|Xs],Extra]).
This code prints:

==========process_join==========
reply for conversation-id prova:1: [a,1],corr_a
reply for conversation-id prova:1: [b,1],corr_b
Joined join_1 for XID=prova:1 with inputs: [[b,1], [c,1]]
reply for conversation-id prova:1: [c,1],corr_c
Joined join_2 for XID=prova:1 with inputs: [[c,2], [d,2]]
reply for conversation-id prova:1: [d,1],corr_d
```

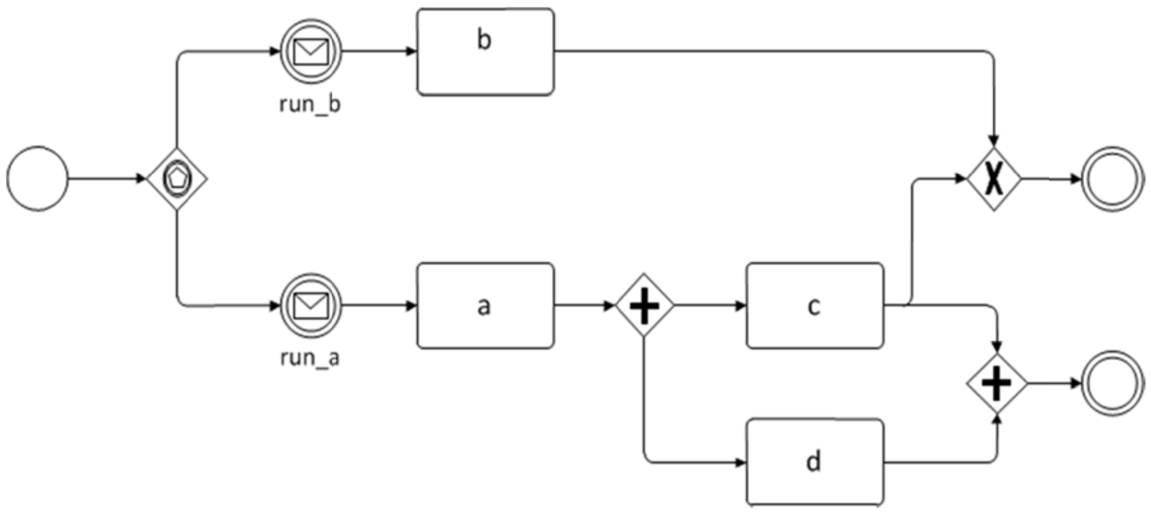
In the example, all tasks are modeled as messages sent to *self*, i.e., the engine itself, and accepting the same message in the body of the same rule. This could be replaced by either in-body task execution, spawning a computation using spawn or sending a request message to an agent and accepting a reply with the results. Remember that rcvMsg does not block the current thread but waits for a message asynchronously, while keeping all the current data in the transparently created closure. Parallel branching is implemented by simply creating a predicate with multiple clauses and processing the relevant tasks asynchronously.

Now back to the *init_join* and *join* predicates. The former initializes a join (an inclusive gateway) by passing it (1) the conversation-id on which the join will be processed, (2) the exit predicate invoked when the join conditions are verified and (3) a list of *join terms* representing tokens that need to become available for the exit predicate of the join to fire. It is entirely possible to specify *join terms* containing arbitrary free variables, including anonymous variable '_'.

When processing reaches a point in the workflow where the join condition can be communicated to the engine, the code uses the *join* predicate that accepts the name of the agent, the conversation-id, the name of the join, and the concrete *join term* that corresponds to a newly available join token. When the engine detects that all join terms are available, it processes the *exit goal* for the predicate corresponding to the join name specified at initialization (see clauses for *join_1* and *join_2*).

1.10.2. Event-driven gateways (edBPM)

Prova supports ebPBM-style of agent programming and, in fact, offers much more with the concept of reaction groups. One could argue that an event-driven gateway is a simplified version of a reaction group. Consider the following variation on the example in the previous section.



What changed here is the exclusive choice that the environment imposes on the process depending on whether the event *run_a* or *run_b* arrives first. Moreover, as soon as either of them is detected, the other alternative branch is immediately disabled so that only one branch can proceed.

The following modified first half of the above example shows how the event-driven gateway is implemented using a reaction group.

```
fork_a_b(XID) :-
   @group(g)
   rcvMsg(XID,Protocol,From,command,run_a),
   sendMsg(XID,self,0,reply,a(1),corr_a),
   rcvMsg(XID,self,Me,reply,a(1),corr_a),
   fork_c_d(XID).
fork_a_b(XID) :-
   @group(g)
   rcvMsg(XID,Protocol,From,command,run_b),
   sendMsg(XID,self,0,reply,b(1),corr_b),
   rcvMsg(XID,self,Me,reply,b(1),corr_b),
   % Tell the join join_1 that a new pattern is ready
   join(Me,XID,join_1,b(1)).
fork_a_b(XID) :-
   @or(g)
   rcvMsg(XID,Protocol,From,or,_).
```

An OR reaction group gives the two message handlers for events *run_a* and *run_b* a collective behavior such that the group collectively waits for either of the two events to arrive. When either event arrives, due to the semantics of OR**,** the group as a whole is terminated so that whatever happens to be the alternative reaction, disappears. The @or annotated reaction is the exit channel from the group that is used for defining the semantics of the group as a whole. In this instance, we are actually not interested in common action logic so this reaction is not followed by anything else. However, there are sub-branches following either of the reactions annotated with @group (this annotation declares them to be part of the reaction group with a logical name *g*. So once *run_a* or *run_b* arrives, the group as a whole disappears but the closure of the fired branch continues as the only extension of the workflow. Note that we use a Prova OR reaction group to model an XOR situation.

The beauty of this approach is the **linearity** of extensible semantic variants that can be added to this specification, translating to the ease of authoring and maintainability of the design, be it workflow, event pattern, or more general reactive behavior. For example, the following modification imposes a 10 seconds timeout on the group, adds a @not annotation on the *run_b* channel and a timeout channel.

```
fork_a_b(XID) :-
```

```
    @group(g)
    rcvMsg(XID,Protocol,From,command,run_a),
    sendMsg(XID,self,0,reply,a(1),corr_a),
    rcvMsg(XID,self,Me,reply,a(1),corr_a),
    fork_c_d(XID).
fork_a_b(XID) :-
    @group(g) @not
    rcvMsg(XID,Protocol,From,command,run_b).
fork_a_b(XID) :-
    @or(g) @timeout(10000)
    rcvMsg(XID,Protocol,From,or,_).
fork_a_b(XID) :-
    @or(g)
    rcvMsg(XID,Protocol,From,timeout,_),
    sendMsg(XID,self,0,reply,b(1),corr_b),
    rcvMsg(XID,self,Me,reply,b(1),corr_b),
    % Tell the join join_1 that a new pattern is ready
    join(Me,XID,join_1,b(1)).
```

If *run_a* arrives before the timeout, it remains the only branch as in the previous case. If neither *run_a* nor *_run_b* arrive before the timeout, the timeout channel proceeds with the branch with task *b*. Prova 3.0 includes a large collection of annotations for reaction groups that really help with designing very sophisticated workflows.

1.10.3. Predicate gateways

It is easy to see why simple joins may be quite limiting. Imagine you need to analyze what join tokens are available to the join and make a decision using logical reasoning. Furthermore, if the final join conditions are not yet achieved, it would be great if we could re-initialize the join, if some other conditions are verified. There are known workflow patterns (for example, Structured Discriminator) cataloged by van-der-Aalst that benefit from such features.

The following example *predicate_join_concurrent.prova* uses the enhanced version of the same workflow framework complemented by built-in predicates *init_predicate_join* and *predicate_join*.

```
:- eval(predicate_join_concurrent()).

predicate_join_concurrent() :-
    println(["==========predicate_join_concurrent=========="]),

    for([0,2],I),
    sendMsg(XID,async,0,request,a()),
    rcvMsg(XID,async,Me,reply,a()),

    % Initialise the predicate join:
    %    XID is conversation id;
    %    join_1 is the exit predicate that must be unique for each join;
    %    join_predicate_1 is the join conditions;
    %    i1..i3 are the required tokens.
    init_predicate_join(XID,join_1,[i1(),i2(),i3()],join_predicate),
    println([XID]),
    % This will create three parallel processing streams.
    loop_a_body("Worker",XID).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Three branches executing three parallel activities %
% representing a divergence in the process           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

loop_a_body(Partner,XID) :-
```

```
    switch_thread(a),
    println(["Branch 1 complete"]),
    predicate_join(XID,join_1,i1(),[Partner]).
loop_a_body(Partner,XID) :-
    switch_thread(b),
    println(["Branch 2 complete"]),
    predicate_join(XID,join_1,i2(),[Partner]).
loop_a_body(Partner,XID) :-
    switch_thread(c),
    println(["Branch 3 complete"]),
    predicate_join(XID,join_1,i3(),[Partner]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Rules representing the exit branches of the JOIN %
%%%%%%%%%%%%%%%%%%%%%%%%%%%

% The following rule will be invoked by join once ONE input for join_1 has arrived.
% This is governed by the first join_predicate rule.
join_1(s(0),XID,Count,Inputs,[Partner]) :-
    println(["Joined for XID=",XID," at state 's(0)' with inputs: ",Inputs]),
    % Report to the test case runner that this step is complete
    sendMsg(XID,self,0,job_completed,[]).
% The following rule will be invoked by join once all inputs for join_1 have arrived
% This is governed by the second join_predicate rule.
join_1(reset,XID,Count,Inputs,[Partner]) :-
    println(["Joined for XID=",XID," at state 'reset' with inputs: ",Inputs]),
    sendMsg(XID,self,0,body_a_complete,[]),
    % Report to the test case runner that this step is complete
    sendMsg(XID,self,0,job_completed,[]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Rules establishing when the predicate join reaches appropriate states %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% The rule for determining a join condition. Note that the state must have format "s(Number)".
join_predicate(XID,join_1,s(0),Msg,Waiting,Count,Complete,Params) :-
    1=Complete.size().
% The rule for determining a reset condition. When this condition is reached, the JOIN is reset.
% However, we also offer a chance to define an optional JOIN exit branch when the reset is reached
%    (see rule 2 for the join_1 predicate).
join_predicate(XID,join_1,reset,Msg,Waiting,Count,Complete,Params) :-
    0=Waiting.size().

switch_thread(A) :-
    sendMsgSync(XID,task,0,switch,[A]),
    rcvMsg(XID,task,From,switch,[A]).

%%%%%%%%%%
% A NOOP worker %
%%%%%%%%%%

rcvMsg(XID,Protocol,From,request,[X|Xs]) :-
println([rcvMsg(XID,Protocol,From,request,[X|Xs])]),
    sendMsg(XID,Protocol,0,reply,[X|Xs]).

% A testing harness catch-all reaction for printing all messages.
rcvMsg(XID,Protocol,From,Performative,[X|Xs]) :-
    println([Performative,[X|Xs]]).
```

```
% A testing harness catch-all reaction for printing all messages.
rcvMsg(XID,Protocol,From,Performative,[X|Xs],Extra) :-
   println([Performative,[X|Xs],Extra]).
```

1.11. Functional programming extensions

Functional programming is all about being able to compose functionality. This ability is related to the glorious category theory that emphasizes maps (*arrows*) over objects. Being slightly controversial, I would claim that functional programming languages should primarily strive for that simplicity to compose, i.e., linearly augment things, rather than getting bogged down in difficult to read syntax and detailed refinement.

The Prova extension for functional programming attempts to do away with the proliferation of syntactical constructs and distill that easy composability. What has been done so far is by no means complete but it offers a number of immediately useful features such as

- single- and multi-valued functions, the latter offering direct support for non-determinism and backtracking;
- functional composition with the extended *derive* built-in;
- partial evaluation;
- lambda functions;
- monadic functions;
- monadic *bind* using a composition of *map* and *join*;
- *maybe*, *list*, *state*, *tree* and *fact* monads as part of the provided library with easy extensibility.

In order to utilize the functional extensions in Prova, you need to consult the utility library *functional.prova* from your rulebase.

```
% This assumes the indicated path location for the library
:- eval(consult('rules/reloaded/functional.prova')).
```

From the source code Prova distribution, you can also run the tests in *ProvaFunctionalProgrammingTest.java* to find out what is available.

1.11.1. Representing functions

Prova allows the user to define functions using facts and rules obeying a number of simple conventions.

1. Functions are defined via rules for arity 2 predicates whose predicate symbol then becomes the function symbol.
2. The first argument is the input and the second argument is the output of the function.
3. If the input (output) is a simple term, this corresponds to a one input (output) parameter function.
4. Otherwise, the input (output) is a Prova list, corresponding to multiple input (output) parameters.
5. The only exceptions to rules 4 and 5 are Prova lists beginning with strings *maybe*, *list*, *state* or *tree*, in which case those are treated as a single input or output parameter.

Consider some examples.

```
double(X,XM) :-
   XM=2*X.

add([A,B],Z) :-
   Z=A+B.

putState([V,S],state([V,V])).
```

```
plusminus(X,X).
plusminus(X,XM) :-
   XM=-X.
```

The first two of them should be self-explanatory while the third generates a state monad representation with equal value and state. The last example is non-deterministic and defines a multi-valued function that takes a number and either echoes it back or returns its negation.

1.11.2. Functional composition

Remember that everything about functions is about composition. Prova currently offers two mechanisms for composing functions: *simple composition* and *monadic composition*.

In both cases, Prova currently requires a built-in predicate *derive* to be used for executing functional pipelines. The functional flavor of *derive* takes a single parameter, which is a Prova list with exactly three elements:

1. functional pipeline,
2. input,
3. output.

Let us see some examples.

```
% Instantiates L1=6
derive([double,3,L1])

% Instantiates L2=4
derive([[add(1)],3,L2])

% Instantiates L3=8
derive([[double,add(2)],3,L3])
```

The *functional pipeline* is represented by a single function (*double* in the first example), a partially evaluated function (*add(1)* in the second example, note the required square brackets around it), a lambda function described later, or a list of those, executed in sequence (*double,add(1)* in the last example).

The input and output can be pretty much any simple or compound Prova terms. In the case when output is specified, Prova will unify the results with the provided pattern, only succeeding if there is a match, which allows for easy constraint solving using normal backtracking if there is non-determinism encountered along the way, perhaps due to multi-valued functions.

1.11.3. Lambda functions

We need one more building block to fully appreciate functional composition, viz., *lambda functions*. Lambda functions can be used directly in function pipelines for specifying a function right there as opposed to using pre-defined functions defined using rules. One advantage is immediate flexibility and the ability to define and pass around lambda expressions even between distributed agents. A more direct advantage is the ability to obtain access to data available from the previous stage of the pipeline and use them precisely in the lambda expression, as opposed to the default splicing of that data at the end of the current function arguments.

The syntax is simple:

```
lambda(Var,Function)
```

Here *Var* is a variable that gets instantiated to the data coming from the input. *Function* is again either a single (possibly, partially evaluated) function or a list of functions representing a nested functional pipeline. Note that *Function* is not required to include *Var*. Note that lambda variables are visible and can be used in all subsequent functions in the enclosing pipeline which is a great way to pass the data downstream.

```
% Two equivalent forms, without and with lambda
% Both instantiate L2=4
derive([[add(1)],3,L2])
derive([[lambda(A,add(1,A))],3,L2])

% An example of passing data downstream
% This instantiates: V3=9, V2=4, V1=2, C1=13
derive(
   [[lambda(V1,double(V1)),lambda(V2,double(3)),lambda(V3,add(V2,V3))],2,C1]
)
```

1.11.4. Simple functional composition

In the case of simple composition, the functional pipeline is composed of functions operating on the totality of the values being passed between them. To appreciate the power of functional composition, we would like to pass between functions compound data as opposed to single values. Consider the following function.

```
duplicate(X,[X,X]).
```

Whatever the *X* value, the function creates a two-element list with two identical copies of the same value (or structure, if *X* is also a list). Following is an example of its use.

```
% Returns XX=6
derive(
   [[duplicate,add],3,XX]
)
```

What happens here? First, the simple value 3 is transformed into the list [3,3]. Then these two numbers (3 and 3) get shipped as the first two parameters to the function *add*, returning the result. The apparent inflexibility of always attaching the computation results to the end of the next function's arguments is nicely resolved by using lambda functions.

```
% Given this definition
second_degree([A,B,C,X],R) :-
   R=A*X*X+B*X+C.

% Splice 3 as both A and B
% This instantiates: Z=3, XX=7
derive(
   [[lambda(Z,second_degree(Z,Z,1,1))],3,XX]
)
```

Now finally if the input is a list, the lambda function needs to accept more than one variable by declaring its input as a list.

```
% This instantiates: A=3, B=3, L11=7
derive(
   [[duplicate,lambda([A,B],second_degree(A,B,1,1))],3,L11]
)
% This also works the same way
derive(
   [[lambda([A,B],second_degree(A,B,1,1))],[3,3],L11]
)
```

Now we are completely ready for FizzBuzz, functional Prova style. We are going to show three ways to do the same thing and later do the same again using the List monad. Note that *for* is a multi-valued function that non-deterministically returns all integers between 1 and 100 (its definition is very simple and is part of the utility library *functional.prova*). prtln is a function that echoes back its input while printing it followed by newline.

```
:- solve(
   derive(
       [[for(1,100),fizzbuzz(3,5),prtln],[],X]
   )
).
:- solve(
   derive(
       [[for(1,100),lambda(I,fizzbuzz(3,5,I)),prtln],[],X]
   )
).
:- solve(
   derive(
       [[for,lambda(I,fizzbuzz(3,5,I)),prtln],[1,100],X]
   )
).

% fizzbuzz: (M1,M2,I)->R
fizzbuzz([M1,M2,I],R) :-
   C1 = I mod M1,
   C2 = I mod M2,
   fizzbuzz(C1,C2,I,R).

fizzbuzz(0,0,_,fizzbuzz) :- !.
fizzbuzz(0,_,_,fizz) :- !.
fizzbuzz(_,0,_,buzz) :- !.
fizzbuzz(_,_,I,I).
```

1.11.5. Monadic functional composition

In the case of monadic composition, the functional pipeline passes around monadic data and is composed of functions operating on data in some form "contained" in the monadic data. Monadic composition in Prova is done by pattern-matching the data passed between functions in the functional pipeline.

To clarify, this approach is not directly comparable to the one used in functional programming languages. The latter is based on strong typing enforced at compile-time. Prova is obviously quite a bit more dynamic so the approach used here is data-driven. In the following discussion, we assume minimal understanding of what monads are about (see, for example, this compact post: http://www.haskell.org/haskellwiki/Monads_as_containers).

The core idea is that monadic functions produce Prova terms matching pre-defined patterns known to the functional framework.

The *Maybe* monad

This monad is represented by Prova terms (recall that Prova terms are represented as Prova lists with the first element equal to the term symbol) that match either of these two patterns:

```
maybe(nothing())
maybe(just(SomeData))
```

It captures the intuition of a computation that either produces a result or returns a “no-result”—for example, a NaN value resulting from a division by zero. Now consider the following monadic function (halve_f:: a->Mb in Haskell):

```
halve_f(0,maybe(nothing())) :- !.
halve_f(Number.A,maybe(just(Number.B))) :-
   Number.B=Number.A/2.
```

This function halves a number but returns a *Nothing* if the input is 0. The following monadic functional pipeline halves a number a few times, eventually resulting in *Nothing*.

```
% This returns: X=[maybe,[nothing]]
:- solve(
   derive(
        [[map(halve_f),join,map(halve_f),join,map(halve_f),join,map(halve_f),join],maybe(just(7)),X5]
    )
).
```

The pipeline is composed of (join . map(f)) paired compositions. The *map* function opens monadic data and operates on its contents (the payload inside *just(…)*). This is mathematically represented as Ma->MMb function in Haskell. The *join* function then unwraps a double wrapped result: join: MMb->Mb so the final result is again in the *Maybe* monad form and can be shipped to the next computation stage. The composition of map and join is known as *bind* and is the key combinator allowing the monadic functions f: a->Mb to be composed into Ma->Mb thereby allowing for such transformations to continue.

The *List* monad

The list monad is represented as a Prova list of the form: *[list,e1,e2,…,en]*, which is the same as *list(e1,e2,…,en)*. The monadic functions for lists operate on individual elements and produce lists (mapping simple data to monadic data). For example, the following function duplicates an element.

```
duplicate_m(X,list(X,X)).
```

Here is a pipeline that uses map followed by join (which is the concatenation operation for lists).

```
% In Haskell: [1,2,3] >>= duplicate_m >>= duplicate_m
%   this returns X3=[list,1,1,1,1,2,2,2,2,3,3,3,3]
:- solve(
    derive(
        [[map(duplicate_m),join,map(duplicate_m),join],list(1,2,3),X3]
    )
).
```

This is an example of a function that filters a list elements based on the filter function.

```
gt_m([X,A],list(A)) :-
    A>X,
    !.
gt_m([X,A],list()).
```

The above function wraps the input number *A* if *it* is greater than *X*, otherwise, it returns an empty list. Now filtering a list is as simple as:

```
% This returns X0=[list,3,4]
:- solve(
    derive(
        [[map([[gt_m(2)]]),join],list(1,2,3,4),X0]
    )
).
```

The *State* monad

The state monad is represented as a Prova list of the form: *[state,[Value,State]]*, which is the same as *state([Value,State])*. It captures the intuition of a function that not only transforms its input *Value* but also causes a change in its environment, represented by the *State* variable. The state monadic functions never change the initial state, instead they output a new value for the state.

Consider the following state monadic function. It flips the state and converts the input to uppercase if the initial state is true, or to lowercase, otherwise.

```
flipcase([C,true],state([CM,false])) :-
    CM=C.toUpperCase().
flipcase([C,false],state([C,true])) :-
    CM=C.toLowerCase().
```

Now execute the following pipeline.

```
% state(['a',true]) >>= (\x -> flipcase x) >>= (\y -> flipcase y)
% This returns: X10=[state,['a',true]]
:- solve(
    derive(
    [[map(flipcase),join,map(flipcase),join],state(['a',true]),X10]
    )
  ).
```

We recover the original value and state after executing two flips, which is not surprising. The State monad, of course, has a lot of uses that we are not discussing here.

The *Fact* monad

Prolog-like languages typically have problems dealing with their fact bases, in particular, as updating the existing facts is problematic. Retracting a fact and adding a new version (a) may change its position among the facts for the same predicate and (b) it is not efficient.

Prova makes use of the functional approach to provide an **in-place fact editing**. For this to work, we introduce monadic terms of the form: *[fact,[Pred|Args]]* that encapsulate computations on facts (see the examples in *func_fact_monad.prova*). In order to modify the existing facts, one writes a monadic fact transformer function that takes a term stored in a fact (along with any additional data) and generates an updated fact term. The system then replaces the original fact on the spot. Note that the transformer function itself is a pure function. Internally it does not change the fact but just produces a new fact term.

The following example demonstrates how we can use the Fact monad.

```
% Some facts
data(1,2).
data(2,3).
data(3,2).

% A fact transformer
inc_snd(A(X,Y),A(X,YM)) :-
  YM=Y+1.

% A pipeline
update_1(R) :-
  derive(
     [[map(inc_snd)],fact(data(X0,2)),R]
  ).
```

Executing *update_1* above will increment the second argument of each fact for predicate d*ata* that has 2 in the second position (the facts *data(1,2)* and *data(3,2)* are replaced with *data(1,3)* and *data(3,3)*, respectively).

Combining monads

The presented data-driven monadic approach is well suited for combining different monads. It only sounds too complicated but in fact, is easy to understand and use. This is a real problem: we want to modify the facts (using the Fact monad) but also to count the number of occurred modifications. Obviously, we want to use pure (side effect-free) functions so global variables is no the solution we are looking for.

Here is the upgrade of the previous example.

```
data(1,2).
data(2,3).
data(3,2).

add_snd_st([D,F(X,Y),S],[F(X,YM),SM]) :-
  YM=Y+D,
  SM=S+1.

update_st2(R) :-
  derive(
     [[ map( [map(lambda(A,add_snd_st(1,A)))] ) ],state([fact(data(X0,2)),0]),R]
  ).
```

This works by nesting a fact monad term inside a state monad term and passing this as the input to the pipeline. The function in the pipeline is a nested *map* that applies the supplied lambda function to the fact terms *F(X,Y)* as they are discovered by the matcher, along with auxiliary data *D* and the current state variable *S*. Both the fact term and the state variable then are transformed to new values. The pipeline that inspects the monadic data takes care of the rest: it threads the state to the possible next stages of the pipeline and updates the fact on the spot before iterating to the next fact matching the supplied pattern *data(X0,2)*.

Stream fusion and unfoldr

Functional pipelines have a lot of value in terms of conceptual clarity and, in the monadic case, also hide appropriate combinators that are based on the types of the data passed between stages. However, if one compares what actually happens when the standard pipeline is run, one will see (this is true not only in Prova but also Scala or Haskell) that the full containers (for example, lists) get created at each stage of the pipeline. This is a lot of waste and is clearly different from the way an imperative program with loops typically would do. The latter operate on individual data one at a time passing it along the pipeline and then iterating to the other contained data.

The technique called stream fusion allows one to write functional pipelines the usual way but fuse the transformations and execute them iteratively, producing one final result at a time.

The function *unfoldr* is a hidden gem in the Haskell arsenal that builds streams from a seed and a step function, outputting results one by one as they are produced. It is close in spirit to stream fusion and in our approach, can be used together with other functional transformations operating on lists or streams.

Consider a Scala function:

```
def scalaLibrarySum(a : Array[Int]) = a.map(i => i * 3 + 7).filter(i => (i % 10) == 0).foldLeft(0)(_ + _)
```

This is what we want to compute but the way Scala does it, creating full intermediate lists, is not *how* we will do it. This is our code (see the example *func_010.prova*).

```
% This returns:
% SF1=[state,[[list,10,40],50]]
:-solve(stream_fusion_1(SF1)).

a([I,S],[IM,S]) :-
   IM=I*3+7.

% The returned list(I) prepends I to the result list
b([I,S],[list(I),SM]) :-
   0=I mod 10,
   !,
   SM=S+I.
% The returned list() allows I to be skipped in the result list
b([I,S],[list(),S]).

stream_fusion_1(X) :-
   derive(
        [[map([map([a,b])])],state([list(1,-1,3,11),0]),X]
   ).
```

The input to our pipeline is a list *list(1,-1,3,11)* and a seed 0 wrapped in a State monad term. We map a composition of functions *a* and *b* to the list elements. The function *a* transforms the supplied value while keeping the state. The function *b* wraps the result value in a list monad term and increments the state variable if the value passes a test. Otherwise, the state stays the same and the generated list term is empty. The net result of running this is that the transformed elements are included in the generated list and the state is the cumulative sum of these terms.

Internally, this works very much like a pure iteration with recursion unfolded and no intermediate structures created, apart from the results used for one input element at a time.

The example *func_012.prova* shows how you can operate on more than data structure at a time, in this instance, zipping the two lists, while running it with a state monad. It is as simple as passing two lists inside a State monad on input and redefining the per-element functions to work on pairs of data.

```
% This returns:
% SF2=[state,[[list,10,10,370],390]]
:-solve(stream_fusion_2(SF2)).

a([[I,J],S],[[IJ,IM],S]) :-
   IJ=I*J,
   IM=IJ*3+7.

b([I,S],[list(I),SM]) :-
   0=I mod 10,
   !,
   SM=S+I.
b([IJ,S],[list(),S]).

stream_fusion_2(X) :-
   derive(
      [[map([map([a,lambda([[IJ,IM],S],b(IM,S))])])],state([[list(1,-1,3,11),list(1,-1,3,11)],0]),X]
   ).
```

The following final example *func_012.prova* shows how a Fibonacci series can be constructed using what closely resembles the *unfoldr* function.

```
% This returns:
% SF1=[state,[[list,1,2,3,5],[]]]
:-solve(test_unfoldr_1(SF1)).

fibs([[],maybe(just([I,J]))],maybe(just([J,K]))) :-
   K=I+J.

until([N,maybe(just([I,N]))],[list(N),maybe(nothing())]) :-
   !.
until([N,maybe(just([I,J]))],[list(J),maybe(just([I,J]))]).

test_unfoldr_1(X) :-
   derive(
      [[map([map([fibs,until(5)])])],state([list([]),maybe(just([0,1]))]),X]
   ).
```

The inpt is again a State monad term that wraps a list with an empty element (effectively signalling that we are not consuming elements from any actual list but simply generating a list from other available data) and a Maybe monad term with initial pair of the Fibonacci series. We output the series elements until the output element is equal to the supplied value (5). The function *fibs* does not require any input list elements, effectively operating indefinitely. It generates a new pair of results wrapped in a Maybe monad, in this case, always a valid value (a ***Just***, not a ***Nothing***). This result is shipped to the *until* function that only returns a ***Just*** in the case the result does not pass the completion test. Otherwise, it outputs a ***Nothing***. The pipeline considers the computation to be complete when the state variable becomes a ***Nothing*** and the output contains the expected series (with the exception of the first two elements).

1.12. Builtins reference

This Section provides a reference for the built-in primitives in the Prova language.

assert/1 (i)

This builtin predicate takes a Prova list as the only argument and adds it to the rulebase run by the engine as a fact (a rule without a body) for the predicate symbol corresponding to the first element in the supplied list. Remember that in Prova, a standard Prolog-like compound term *f(t1,...,tN)* is a syntactic equivalent of a list *[f,t1,...,tN]*, so we see that the predicate symbol corresponds to the functor *f* of the compound term. Note that Prova facts can contain free variables.

```
TL2 = javax.swing.JLabel("Results"),
TL2.setAlignmentX(java.awt.Component.CENTER_ALIGNMENT),
assert(label2(TL2)),
```

The asserted facts can be queried in the same way as the regular facts that were part of the rulebase at the time the Prova engine started.

```
label2(TL2),
unescape("\ntest_spawn 2 finished.",Text),
```

The asserted facts can removed using retract/1 or retractall/1 builtin predicates.

```
Comp = Event.getComponent(),
% Remember where the popup occurred
retractall(selected_component(_)),
assert(selected_component(Comp)),
```

asserta/1 (i)

This builtin predicate takes a Prova list as the only argument and adds it **in front of any other facts for the same predicate** to the rulebase run by the engine as a fact (a rule without a body) for the predicate symbol corresponding to the first element in the supplied list. Remember that in Prova, a standard Prolog-like compound term *f(t1,...,tN)* is a syntactic equivalent of a list *[f,t1,...,tN]*, so we see that the predicate symbol corresponds to the functor *f* of the compound term. Note that Prova facts can contain free variables.

The following example demonstrates.

```
:- solve(test008(X,Y)).

test008(X,Y) :-
    assert(symmetric(f)),
    % The fact below will be added higher than the one above
    asserta(symmetric(g)),
    symmetric(X),
    println(["Rule A1: symmetric",X]," "),
    symmetric(Y),
    println(["Rule A2: symmetric",Y]," ").
test008(X,Y) :-
    assert(dual(h,a)),
    assert(dual(h,_)),
    asserta(dual(h,f)),
    retractall(dual(_,a)),
    dual(X,Y),
    println(["Rule B: dual",X,Y]," ").
```

The example prints:

```
Rule A1: symmetric g
Rule A2: symmetric g
X=g, Y=g
Rule A2: symmetric f
X=g, Y=f
```

```
Rule A1: symmetric f
Rule A2: symmetric g
X=f, Y=g
Rule A2: symmetric f
X=f, Y=f
Rule B: dual h f
X=h, Y=f
Rule B: dual h <50>
X=h, Y=<50>
```

attach/3 (i,i,io)

This predicate appends the Prova list in the second argument to the Prova list in the first argument, returning the result in the third. Since Prova lists are held in arrays, the code can concatenate very large lists.

In the first argument,

- the list can only have a tail that is a list--not a free Prova variable,
- the tail list is allowed to have a tail but this tail is ignored and not copied to the result List.

In the second argument,

- the list can only have a tail and the tail is attached to the end of the result list.

Here are some examples.

```
:-solve(attach([1|L],[2,3],X)).

:-solve(attach([1|[1,L]],[2,3],X)).

:-solve(attach([1|[1|L]],[2,3],X)).

:-solve(attach([1|[1|L]],[2,3|Z],X)).
```

This prints:

```
no
L=L, X=[1,1,L,2,3]
L=L, X=[1,1,2,3]
L=L, Z=Z, X=[1,1,2,3|Z]
```

Here is a classic example of tower of Hanoi puzzle in Prova, also making use of Prova tabling using the cache built-in predicate.

```
:-solve(move(4,left,mid,right,Plan)).

move(0,_,_,_,[]):-!.
move(N,A,B,C,Plan) :-
   NM1=N-1,
   cache(move(NM1,A,C,B,PlanL)),
   cache(move(NM1,B,A,C,PlanR)),
   attach(PlanL,[[A,C]|PlanR],Plan).
```

bound/1 (i)

This predicate fails if the supplied argument is a free Prova variable or a Prova list containing Prova variables. Otherwise, it succeeds.

The following fragment shows how to test if the argument supplied to the clause is bound.

```
access_data(Type,ID,Data,CacheData) :-
   % Attempt to retrieve bound data
   bound(ID),
   Data=CacheData.get(ID),
    % Success, Data (whatever object it is) is returned
    !.
```

byte_stream/2 ([i,i],o)

This predicate encodes either an input string or a Java *ByteArrayOutputStream* using a supplied charset name (see IANA Charsets) and returns a Java *ByteArrayInputStream* wrapping the resulting byte array. The predicate is structured as a Prova function (see Functional programming extensions). In the first argument, it accepts a Prova list with two input parameters: (1) an input string or a *ByteArrayOutputStream* and (2) a charset name. The second argument is the produced *ByteArrayInputStream*.

The following example compresses a Java *String*, stores to disk, and retrieves it back.

```
:- solve(test_byte_stream(toto)).

test_byte_stream(Result) :-
    println(["==========byte_stream=========="]),
    byte_stream(["toto","UTF-8"],BAIS),
    File=java.io.File.createTempFile("prefix","suffix"),
    FO=java.io.FileOutputStream(File),
    ZFO=java.util.zip.GZIPOutputStream(FO),
    copy_stream(BAIS,ZFO),
    println(["Compressed file created."]),

    FI=java.io.FileInputStream(File),
    ZFI=java.util.zip.GZIPInputStream(FI),
    BAOS=java.io.ByteArrayOutputStream(),
    copy_stream(ZFI,BAOS),
    byte_stream([BAOS,"UTF-8"],Result),
    println(["The original string read from the compressed file: ",Result]).
```

The following test from *ProvaBuiltins1Test.java* shows how this is run from Java.

```
@Test
public void byte_stream_and_copy_stream() {
    final String rulebase = "rules/reloaded/byte_stream.prova";
    final int[] numSolutions = new int[] {1};

    ProvaCommunicator prova = new
ProvaCommunicatorImpl(kAgent,kPort,rulebase,ProvaCommunicatorImpl.SYNC);
    List<ProvaSolution[]> solutions = prova.getInitializationSolutions();
    org.junit.Assert.assertEquals(1,solutions.size());
    org.junit.Assert.assertEquals(numSolutions[0],solutions.get(0).length);
}
```

cache/2 (i)

This builtin predicate provides support for tabling in Prova. It enables the literal in its only argument for caching. The current implementation in Prova 3.0 keeps the cached answer set for the literal indefinitely.

The following example runs the Hanoi Tower puzzle for 20 disks, which would have been prohibitively long if the *cache* predicate was not used. Naive implementations in Haskell or Scala also suffer unless tabling is explicitly added.

```
:-solve(move(20,left,mid,right,Plan)).

move(0,_,_,_,[]):-!.
move(N,A,B,C,Plan) :-
    NM1=N-1,
    cache(move(NM1,A,C,B,PlanL)),
    cache(move(NM1,B,A,C,PlanR)),
    attach(PlanL,[[A,C]|PlanR],Plan).
```

Tabling also helps to avoid infinite looping in code as the example below shows.

```
:- solve(reach(X)).

edge(a,b).
edge(c,d).
edge(d,c).

reach(a).
reach(A) :-
   edge(A,B),
   cache(reach(B)).
```

This returns only one solution *X=a*.

capture_enum/2 ([i,i],io) ND

This non-deterministic predicate enumerates regular expression groups. It is structured as a multi-valued Prova function (see Functional programming extensions). In the first argument, it accepts a Prova list with two input parameters: an input string and a regular expression. It produces independent solutions, one each for each possible answer in the second argument. This creates a choice point with as many non-deterministic branches as there are tokens in the input stream.

The following example shows how this works.

```
:- solve(test017(Groups)).

test017(Groups) :-
   Text="A doodle is a doddle",
   % non-deterministically enumerate regular expression groups
   capture_enum([Text,"(d?)(dl)"],Groups).
```

This returns:

```
Groups=[,dl]
Groups=[d,dl]
```

clone/2 (i,io)

This builtin predicate implements a "unidirectional unification". It creates fresh variables for the first term and then unifies it with the second supplied term so that the result is available in the second argument while the first argument is left unmodified. For example,

```
:- solve(test_clone(must_match,add)).

test_clone(B,FunX) :-
   clone(lambda(A,[add,A,4]),lambda(1,[FunX|ArgXs])),
   % At this point, A is still a free variable, but FunX is bound to a string 'add'
   println([FunX,ArgXs,B]," "),
   println(lambda(A,[add,A,4])," "),
   % must_match unifies with a constant A successfully
   % Not we cannot use A as the head variable, otherwise clone cannot match must_match with 1
   B = A.
```

The code above prints the following.

```
add [1,4] must_match
lambda <12> [add,<12>,4]
yes
```

One use of this builtin is to map lambda expressions over lists (and other monads). The copy of the lambda function needs to be fresh on application to each list element, so *clone/2* achieves this by unifying the provided lambda expression in the first argument with a standard template in the second argument, applies the result in the second argument to the current element using the builtin *derive/3* and continues processing the remainder of the list with the original unmodified lambda expression.

```
map([lambda(A,[Fun|Args]),list(X|As)],[B|Bs]) :-
   !,
   clone(lambda(A,[Fun|Args]),lambda(X,[FunX|ArgXs])),
   derive([FunX,ArgXs,B]),
   map([lambda(A,[Fun|Args]),As],Bs).
```

concat/2 (i, io)

This builtin predicate takes a Prova list and iterates over the list elements, by converting them to Java strings, concatenating them, and unifying the result with the second argument. If the second argument is not a free variable, the unification between the concatenation result and this argument may fail which then fails the builtin.

```
% No solutions here
r(AB) :-
   concat([a,b], AB),
   concat([a,c], AC),
   concat([a,c], AB),
   % This line is (correctly) unreachable in Prova 3.0
   println(["1: ",AB," = ",AC]).
```

copy/2 (i,i)

This predicate takes the character stream from a Java *Reader* and writes it using a Java *Writer*. IO exceptions are wrapped in a *RuntimeException*.

The following code uses a *StringWriter* to capture the stream in a buffer. A mobile agent functionality is implemented by sending a rulebase fragment to another agent that then consults the rulebase dynamically and runs a goal.

```
upload_mobile_code(XID,Remote,File) :-
   fopen(File,Reader),
   Writer = java.io.StringWriter(),
   copy(Reader,Writer),
   SB = Writer.getBuffer(),
   sendMsg(XID,esb,Remote,upload,consult(SB)).
...
rcvMsg(XID,Esb,From,upload,consult(SB)) :-
   iam(Me),
   consult(SB),
   % The clause for worker/2 is defined in the code that the Manager uploads to this worker
   worker(XID,From).
```

copy_stream/2 (i,o)

This predicate copies a Java *ByteArrayInputStream* to a Java *ByteArrayOutputStream*. The predicate is structured as a Prova function (see Functional programming extensions), with the first argument as input and the second argument as output.

The following example compresses a Java *String*, stores to disk, and retrieves it back.

```
:- solve(test_byte_stream(toto)).

test_byte_stream(Result) :-
   println(["=========byte_stream========="]),
   byte_stream(["toto","UTF-8"],BAIS),
   File=java.io.File.createTempFile("prefix","suffix"),
   FO=java.io.FileOutputStream(File),
   ZFO=java.util.zip.GZIPOutputStream(FO),
    copy_stream(BAIS,ZFO),
    println(["Compressed file created."]),

    FI=java.io.FileInputStream(File),
    ZFI=java.util.zip.GZIPInputStream(FI),
    BAOS=java.io.ByteArrayOutputStream(),
    copy_stream(ZFI,BAOS),
    byte_stream([BAOS,"UTF-8"],Result),
    println(["The original string read from the compressed file: ",Result]).
```

The following test from ProvaBuiltins1Test.java shows how this is run from Java.

```
@Test
public void byte_stream_and_copy_stream() {
    final String rulebase = "rules/reloaded/byte_stream.prova";
    final int[] numSolutions = new int[] {1};

    ProvaCommunicator prova = new
ProvaCommunicatorImpl(kAgent,kPort,rulebase,ProvaCommunicatorImpl.SYNC);
    List<ProvaSolution[]> solutions = prova.getInitializationSolutions();
    org.junit.Assert.assertEquals(1,solutions.size());
    org.junit.Assert.assertEquals(numSolutions[0],solutions.get(0).length);
}
```

consult/1 (i)

This builtin predicate dynamically "consults" (imports) a Prova rulebase from either of the following provided as its only input:

- a *String* with file pathname, resource, or URL, in this order.
- a pre-constructed *BufferedReader*.
- a *StringBuffer*.

This predicate is often used in rulebases that need to import another rulebase, typically with some common library rules. This requires running a goal for a meta-predicate *eval* to execute this built-in. Any goals contained in the consulted rulebase are executed synchronously, before returning.

```
:- eval(consult('rules/reloaded/functional.prova')).
```

element/2 (io,i) ND and element/3 (io,io,i) ND

The *element* predicate is a non-deterministic predicate that matches the pattern provided in the first argument with a list of elements. The list can be a Prova list, a Java *List*, or even a Java *Iterator*. Consider this example:

```
% Shows that element/2 can use Java typed and list-based patterns to find matching elements.

:- solve(element_matching([Integer.N,X])).

element_matching([Integer.N,X]) :-
    L = java.util.ArrayList(),
    L.add([X,2]),
    L.add([Integer.I,Double.D]),
    L.add([2.14,Double.D]),
    L.add([3,"toto"]),
    element([Integer.N,X],L).

% The "old" Prova returns:
% <Integer.N>,2
% <Integer.N>,<java.lang.Double.@@9>
% 3,toto
% The correct first solution should be 2,2
```

The 'old' Prova does not unify the elements inside a Java List so it does not correctly unify the pattern [Integer.N,X] with the list element [X,2]. The 'new' Prova correctly matches these elements as they appear inside the Java list (see the final comment in the code above).

Here is an example of extracting one element at a time from a Java Iterator.

```
Iter = Model.listStatements(),
element(Stmt,Iter),
```

The *element/3* version is also returning (or accepting) the index of the matching element. The example below demonstrates.

```
:- solve(element_matching(Index,[Integer.N,X])).

element_matching(Index,[Integer.N,X]) :-
   L = java.util.ArrayList(),
   L.add([X,2]),
   L.add([Integer.I,Double.D]),
   L.add([2.14,Double.D]),
   L.add([3,"toto"]),
   element(Index,[Integer.N,X],L).

% This returns:
% Index=0, N=2, X=2
% Index=1, N=java.lang.Integer.<X71>, X=java.lang.Double.<X72>
% Index=3, N=3, X=toto
```

Implementation details

For this to work, we have added the ability for built-in implementors to create a virtual predicate and add virtual clauses to it. The clauses in the virtual predicate encapsulate alternative non-deterministic choices as possible unification targets for the literal that replaces the current goal as a result of the built-in processing. Hence unification works as normal and correctly matches the *element* pattern with each list element in turn.

fail/0 ()

This predicate results in the immediate subgoal divergence (fail) that results in backtracking to the previous non-deterministic choice point in the goal evaluation tree. Note the Prova syntax for fail that requires parentheses after *fail* like so: *fail()*.

When the following code runs goals *test(X)* of *test(id1)*, it does not return any solutions due to a cut and fail in the first *test* clause. The goal *test(id2)* is successful.

```
test(id1) :-
   println(["Condition id1 being tested."]),
   !,
   fail().
test(id2) :-
   println(["Condition id2 being tested."]),
   !.
```

findall/3 (i,i,o)

This predicate is a customary way in logic programming to accumulate all solutions to a goal passed as a compound term (in Prova, it is just a list). It can be used among other things for returning back all solutions to a goal communicated by a query message back to the requesting agent.

```
% Reaction rule to a general queryref_acc
rcvMsg(XID,Protocol,From,queryref_acc,[ID,[X|Xs]]) :-
   findall([X|Xs],[X|Xs],L),
sendMsg(XID,Protocol,From,reply,[ID,L]).
```

The goal is passed in the second argument and the results are returned in the third argument that currently must be initially a free variable. Normally, The first element *X* of the goal list *[X|Xs]* in the inbound message will be bound to a predicate symbol, for which the solutions are sought. The first argument provides the pattern specifying the actually desired elements to be added to a list given the current goal solution. The call to *findall* instantiates the third argument to be a Java *ArrayList* with all the results.

We could modify the above example like this.

```
rcvMsg(XID,Protocol,From,queryref_acc,[ID,[X|Xs]]) :-
   findall(Xs,[X|Xs],L),
   sendMsg(XID,Protocol,From,reply,[ID,L]).
```

This rule is different in that it does not include the goal predicate symbol, which is always the first element of the list, but only includes the possible instantiations of the goal parameters in the list rest *Xs*.

fopen/2 (i,o)

This predicate returns a *BufferedReader* in the second argument given a file name or a classpath resource in the first argument. If the resource is not found, the built-in will wrap the exception in a *RuntimeException*.

The following code shows how a mobile agent functionality can be implemented by sending a rulebase fragment to another agent that then consults the rulebase dynamically and runs a goal.

```
upload_mobile_code(XID,Remote,File) :-
   fopen(File,Reader),
   Writer = java.io.StringWriter(),
   copy(Reader,Writer),
   SB = Writer.getBuffer(),
   sendMsg(XID,esb,Remote,upload,consult(SB)).
...
rcvMsg(XID,Esb,From,upload,consult(SB)) :-
   iam(Me),
   consult(SB),
   % The clause for worker/2 is defined in the code that the Manager uploads to this worker
   worker(XID,From).
```

for/2 (i,io) ND

The *for* predicate accepts a two-element list with two integers denoting the loop bounds *[From,To]* and non-deterministically enumerates all intermediate values in the second variable that is typically initially free. This predicate is structured in the way compatible with the Prova extensions for functional programming.

Consider this example:

```
   for([1,Max],I),
   sendMsgSync(I,async,0,compute,I)
```

This fragment sends *Max* messages in new conversations with conversation-id equal to *I* running from 1 to *Max*.

free/1 (i)

This predicate succeeds if the supplied argument is a free Prova variable.

The following example *test010.prova* shows how the *free* predicate can be used in th eprocess of constructing a sorted tree representation of an initially unsorted list.

```
% Sort the list provided
:- solve(sort([7,3,6,1,4,5],L)).
```

```
insert_a(Val,t(Val,_,_)):-!.
insert_a(Val,t(Val1,Tree,_)):-
   Val<Val1,
   !,
   insert_a(Val,Tree).
insert_a(Val,t(_,_,Tree)):-
   insert_a(Val,Tree).

instree([],_).
instree([H|T],Tree):-
   insert_a(H,Tree),
   instree(T,Tree).

treemembers(_,T):-
   free(T),
   !,
   fail().
treemembers(X,t(_,L,_)):-
   treemembers(X,L).
treemembers(X,t(X,_,_)).
treemembers(X,t(_,_,R)):-
   treemembers(X,R).

sort(L,L1):-
   instree(L,Tree),
   findall(X,treemembers(X,Tree),L1).
```

This returns:

L=[1, 3, 4, 5, 6, 7]

listen/2 (i,i)

This predicate is part of the Prova extension for Swing. It adds a listener to various Swing events on the specified UI target. The created listener then generates Prova events that can be intercepted with usual Prova reactions. The reactions are executed on the specially designated Swing event thread, not on the usual *task* or *async* thread pools. The *unlisten* built-in predicate removes the appropriate listener.

The built-in accepts the following parameters:

1. the mode - either *action*, *mouse*, *change* or *motion*;
2. the target - either a *java.swing.AbstractButton* for *action* or *mouse* events, or a *java.awt.Component for _change* or *motion* events.

The format of the messages generated by the listener is the same as if it was generated with the following *sendMsg* statements. The class *ProvaSwingAdaptor* is responsible for this mapping.

```
% For action events:
sendMsg(s,task,0,swing,[action,ActionCommand,Source,ActionEvent])
% For state changed events
sendMsg(s,task,0,swing,[change,Source,ChangeEvent])
% For mouse click events
sendMsg(s,task,0,swing,[mouse,clicked,Source,MouseEvent])
% For mouse entered events
sendMsg(s,task,0,swing,[mouse,entered,Source,MouseEvent])
% For mouse exited events
sendMsg(s,task,0,swing,[mouse,exited,Source,MouseEvent])
% For mouse pressed events
sendMsg(s,task,0,swing,[mouse,pressed,Source,MouseEvent])
% For mouse released events
sendMsg(s,task,0,swing,[mouse,released,Source,MouseEvent])
```

You can study the standalone example *swing_rx.prova* that demonstrates how strings of UI actions (in particular, mouse gestures) can be detected as event patterns using Swing events accepted as Prova messages. It follows closely

Here are some critical fragments. First register the listener.

```
   JB1 = javax.swing.JButton("JB1"),...,
   listen(mouse,JB1),
   JB2 = javax.swing.JButton("JB2"),...,
   listen(mouse,JB2),
```

Now detect mouse gestures mapping them to event patterns.

```
detect_gesture1(JB1,JB2) :-
   % Reaction to incoming swing mouse pressed messages.
   rcvMult(s,Protocol,From,swing,[mouse,pressed,JB1,Event]) [1=Event.getButton()],
   println(["Detected mouse press"]),

   detect_drag1(JB1,JB2).

detect_drag1(JB1,JB2) :-
   @group(g1) @not
   rcvMsg(s,Protocol,From,swing,[mouse,pressed,Src,Event])
[E2=javax.swing.SwingUtilities.convertMouseEvent(Src,Event,JB2),P2=E2.getPoint(),Boolean.TRUE=JB2.cont
ains(P2)],
   println(["Detected mouse exited"]),
   detect_drag2(JB1,JB2).
detect_drag1(JB1,JB2) :-
   @group(g1)
   rcvMsg(s,Protocol,From,swing,[mouse,released,Src,Event])
[E1=javax.swing.SwingUtilities.convertMouseEvent(Src,Event,JB1),P1=E1.getPoint(),Boolean.TRUE=JB1.cont
ains(P1)],
   println(["Detected mouse released ",P1]).
detect_drag1(JB1,JB2) :-
   @group(g1) @not
   rcvMsg(s,Protocol,From,swing,[mouse,pressed,Src,Event])
[E1=javax.swing.SwingUtilities.convertMouseEvent(Src,Event,JB1),P1=E1.getPoint(),println(["oho",P1]),Bool
ean.TRUE=JB1.contains(P1)],
   println(["Detected repeated press"]).
detect_drag1(JB1,JB2) :-
   @and(g1)
   rcvMsg(s,Protocol,From,and,Events),
   println(["Gesture detected"]),
```

match/3 (i,io,io)

This predicate is quite unique and is typically used internally in the Prova extension for functional programming (consulted from functional.prova).

What it does can be better illustrated with an example.

```
t(1,2).
t(2,3).
t(3,2).

:-solve(test_match(S)).

test_match([X1,Y1,X2,Y2]) :-
   match(t(X0,2),t(X1,Y1),G),
   match(t(X0,2),t(X2,Y2),G).
```

This example prints:

```
S=[1,2,3,2]
```

The idea behind *match* is that we should be able to iterate over facts (or rule heads) that match the pattern supplied in the first argument. The iteration is executed by repeated invocation as opposed to non-deterministic choice. The second output-only argument represents the actual fact as it is stored in the rulebase rather than the result of its unification with the pattern in the first argument. The third argument must be a free variable on first invocation but becomes instantiated to a special handle that could be compared to an iterator, allowing for subsequent invocations of *match* to return other matching facts. The *match* invocation finally fails when all the matching facts are exhausted.

Moreover, the handle *G* in the above example can be used in the *update* built-in predicate to update the matching fact directly in place, without retracting it and asserting back a modified version.

The functional extension hides *match* and *update* completely, allowing the user to simply define a mutator function that takes the complex term corresponding to a fact literal and produces a mutated version. This function is then passed to the functional pipeline that iterates over the facts and updates them accordingly.

Here is an example.

```
o(1,2).
o(2,3).
o(3,2).

add_snd([F(X,Y),D],F(X,YM)) :-
   YM=Y+D.

f(R) :-
   map([lambda(A,add_snd(A,1)),fact(o(X,2))],R).

:-solve(f(R)).

:-solve(o(S,3)).
```

The fact table *o* undergoes a change that increments its second argument for all facts matching the pattern *o(X,2)*. The second goal proves that all the *o* facts now have 3 in the second position and the original order of the *o* facts is preserved.

```
R=[list,[o,1,3],[o,3,3]]
S=1
S=2
S=3
```

mklist/2 (i,io) ND

The *mklist* predicate assembles ("makes") a Prova list from a rest-free Prova list and a variable representing the list rest. The predicate follows the convention of the functional extensions to Prova to enclose the inputs in a list in the first parameter and produce the resulting list in the second. The first parameter is a list with two lists inside: a Prova list without a tail and a free variable that represents a list tail. It outputs the list composed of the first list together with the supplied rest.

This example shows how this works.

```
:- solve(mklist([[1,2],Xs],Xs)).

% This outputs:
% Xs=Xs, L=[1,2|Xs]
```

parse_list/2 ([i,i],io)

This predicate parses an input string using a supplied regular expression and returns a Prova list with captured tokens. The predicate is structured as a Prova function (see Functional programming extensions). In the first argument, it accepts a Prova list with two input parameters: an input string and a regular expression. The second argument is the produced list of tokens. If this list is supplied as input, it is unified against the output list tokens.

The following example demonstrates.

```
:- solve(test_parse_list("abc:-12a234",L)).

test_parse_list(In,L) :-
   parse_list([In,"(?:(\w*):|)(-?\d*\w?)-?(\d*)"],L).
test_parse_list(In,[T|Ts]) :-
   parse_list([In,"(?:(\w*):|)(-?\d*\w?)-?(\d*)"],[T|Ts]).
test_parse_list(In,L) :-
   parse_list([In,"(?:(\w*):|)(-?\d*\w?)-?(\d*)"],ttt).
```

This test prints the following solutions.

```
L=[abc,-12a,234]
L=[abc,-12a,234]
```

parse_nv/2 ([i,i],[io,io])

This predicate parses an input string with name/value pairs using a regular expression to build an array of names and and an array of values. It is structured as a Prova function (see Functional programming extensions). In the first argument, it accepts a Prova list with two input parameters: an input string and a regular expression. It returns results in another Prova list with two elements: an array of names and an array of values.

The following example shows how this works.

```
:- solve(test_parse_nv_1(Value1)).

:- solve(test_parse_nv_2(Value2)).

test_parse_nv_1(Value) :-
   parse_nv(["j=12,s=tt","(?:(\w+)=(\w+),?)"],[Names,Values]),
   element(Value,Values).

test_parse_nv_2(Value) :-
   parse_nv(["j=12,s=tt","(?:(\w+)=(\w+),?)"],[[N|Ns],[V|Vs]]),
   element(Value,[V|Vs]).
```

This returns:

```
Value1=12
Value1=tt
Value2=12
Value2=tt
```

println/1 (i) or println/2 (i,i)

This builtin predicate accepts a Prova list as the first parameter and prints its elements to the standard out channel using *System.out.println(...)*. The second optional *String* parameter is a separator that is inserted between each element.

The following example demonstrates.

```
:- eval(msg()).

msg() :-
   println(["==========Communicator messaging test 001=========="]),

   % This reaction is active indefinitely
   rcvMult(XID,async,From,inform,{a->I}),
    println(["Received",rcvMsg(XID,async,From,inform,{a->I})],": ").

% Given a stream of inbound messages, this agent prints:
% ==========Communicator messaging test 001==========
% Received: [rcvMsg,prova1,async,0,inform,{a->1}]
% Received: [rcvMsg,prova1,async,0,inform,{a->2}]
% ...
```

rcvMsg/5 (io,io,io,io,io)

This built-in is used inside body rules for receiving messages (see a more in-depth discussion in Using reaction rules for receiving messages). Executing this built-in creates a closure with all the remaining sub-goals in the current goal evaluation. The engine then immediately fails the current branch in the goal exploration and backtracks to the previous choice point. The agent then waits for an inbound message (without holding the current thread) that matches the pattern according to the following arguments.

1. *XID* - conversation id of the message;
2. *Protocol* - name of the message passing protocol;
3. *Sender* - the agent name of the sender;
4. *Performative* - the message type broadly characterizing the meaning of the message;
5. *Payload* - a Prova list containing the actual content of the message.

The *rcvMsg* predicate is by far the most complex in Prova. To appreciate fully what can be done using *rcvMsg*, the Reader will have to familiarize with Reactive messaging and Event processing. *rcvMsg* is used both for accepting individual messages as in Example 1 below and for grouped reactions as in Example 2.

Example 1

The following example *rules/reloaded/simple_async.prova* for each *I* between 0 and 2, inclusive, sends a message over the *async* protocol and receives them on independent threads. sendMsgSync is used for ensuring that messages are sent only after the goal is complete, so that all the subsequent *rcvMsg* are ready to receive messages. *XID* gets instantiated by sendMsgSync each time to a new value as it is initially a free variable.

```
:- eval(simple_async()).

simple_async() :-
    for([0,2],I),

    sendMsgSync(XID,async,0,request,a(I)),
    rcvMsg(XID,async,Me,request,a(I)),
    TH=java.lang.Thread.currentThread(),
    println(["<",XID,"> ",I," on ",TH]).
```

Running it with "prova2.bat simple_async.prova" outputs three lines on random threads, for example:

```
<prova:1> 0 on Thread[pool-8-thread-1,5,main]
<prova:3> 2 on Thread[pool-6-thread-1,5,main]
<prova:2> 1 on Thread[pool-7-thread-1,5,main]
```

Example 2

The following fragment from the flower shop use case in the project mule-prova-agents demonstrates the use of a reaction group to collect exactly *N* initialization messages from *N* drivers with the specified timeout. If all the drivers send the message within the specified timeout, the group proceeds to positive reaction (second clause), otherwise, the timeout is detected and the reaction in the third clause is invoked.

```
% Collect at least one update for each of N drivers prior to proceeding with requests
checkin_drivers(N) :-
```

```
    @group(init_drivers) @count(N,N)
    rcvMsg(VanId,Protocol,VanId,update,gps_coordinates(Latitude,Longitude)).
checkin_drivers(N) :-
    @and(init_drivers) @timeout(10000)
    rcvMsg(VanId,Protocol,VanId,and,[Updates]),
    % All drivers have sent updates within the specified timeout
    % Go-Go-Go! Ready to send orders now
    send_orders().
checkin_drivers(N) :-
    @and(init_drivers)
    rcvMsg(VanId,Protocol,VanId,timeout,[Updates]),
    % Not all drivers have sent updates within the specified timeout, for now just print this
    println(["Not all drivers have sent updates within the specified timeout"]).
```

rcvMult/5 (io,io,io,io,io)

This built-in is a variant of the inline reaction predicate rcvMsg that allows for receiving multiple inbound messages as opposed to a single message as in rcvMsg. As in the case of rcvMsg, the engine does not block the current thread and simply adds the continuation corresponding to the current context goals and variables so that the execution picks up exactly where it left off when the matching message is detected. In the case of *rcvMult*, the original reaction continues processing other messages after the first one is detected.

The predicate is commonly used as an initiator of a workflow or event pattern detection. The example below waits for a creation of a market and then proceeds to further processing in clauses for the predicate *server_1*.

```
server(Limit) :-
    % Start detection on each new market
    rcvMult(Market,Protocol,From,create,market(Market)),
    server_1(Market,Limit).
```

By default, the *rcvMult* inline reaction exists until the Prova agent is terminated. However, there is a way to abandon and purge this reaction.

```
client() :-
    sendMsg(XID,self,0,request,[19]),
    assert(best(XID,0,0)),
    branch(XID).

branch(XID) :-
    rcvMult(XID,self,Me,respond,[Service,Offer]),
    println(["Received offer: ",Offer," from ",Service]),
    choose(XID,BestService,Service,Offer).
branch(XID) :-
    spawn(XID,java.lang.Thread,sleep,1000L),
    rcvMsg(XID,self,Me,return,Ret),
    best(XID,BestService,BestOffer),
    println(["Received best offer: ",BestOffer," from ",BestService]),
    sendMsg(XID,self,0,eof,_).

choose(XID,BestService,Service,Offer) :-
    best(XID,BestService,BestOffer),
    Offer>BestOffer,
    retract(best(XID,BestService,BestOffer)),
    assert(best(XID,Service,Offer)).
```

In the above example from *hohpe_dynamic_discovery.prova*, the client sends a request for proposals with a payload containing some test data (19). It then asserts the record *best* representing the best offer received and in the first *branch* clause receives offers and updates the *best* record. In the second clause, it spawns an asynchronous wait for one second, prints the best offer, and uses a special termination message with command type *eof* that is intercepted by the *rcvMult* reaction that is then terminated.

This example is somewhat contrived as there are better ways to achieve the same result using reaction groups. Here is a solution from *hohpe_using_groups.prova*. The sorted accumulator collects all offers sorted by the offer values. If the *@size(1)* annotation was not added to the exit reaction *@or*, the outputs would have continued every second. The *@size(1)* output just one collection with the results and the exit reaction extracts that first result and prints the highest offer.

```
client() :-
   sendMsg(XID,self,0,request,[19]),
   branch(XID).

branch(XID) :-
   Acc = ws.prova.eventing.SortedAccumulator(),
   @group(bids) @timer('1 sec','1 sec',Acc)
   rcvMsg(XID,self,Me,respond,[Service,Offer]),
   Acc.processAt(Offer,Service),
   println(["Received offer: ",Offer," from ",Service]).
branch(XID) :-
   @or(bids) @size(1)
   rcvMsg(XID,self,Self,or,[Results]),
   Acc=Results.get(0),
   Top = Acc.highest(1),
   println(["Received best offer: ",Top]).
```

read_enum/2 (i,io) ND

This non-deterministic predicate enumerates the lines returned by reading the input *BufferedReader*. It is structured as a multi-valued Prova function (see Functional programming extensions). In the first argument, it accepts a Java *BufferedReader*. The predicate produces independent solutions, one each for each line read. This creates a choice point with as many non-deterministic branches as there are lines in the input.

The following example shows how this built-in predicate works.

```
:- solve(test_read_enum_1(Line)).

test_read_enum_1(Line) :-
   fopen($File,Reader),
   read_enum(Reader,Line).
```

This returns:

```
Line=:- solve(test_read_enum_1(Line)).
Line=
Line=test_read_enum_1(Line) :-
Line=fopen($File,Reader),
Line= read_enum(Reader,Line).
```

retract/1 (i)

This builtin predicate, standard to Prolog-like languages, is the dual of assert/1. It takes a Prova list (which is the same as a compound term in Prova) and unifies it with the facts in the current rulebase until the first match is found. It then removes the matching fact or fails, if it cannot find it. The supplied term can contain variables and unification between these variables and the constants or variables in the fact is done using the usual unification rules. Note that Prova facts are rules without a body and as such can contain free (uninstantiated) variables.

The following fragment will non-deterministically enumerate the *symmetric(g)* and *symmetric(h)* facts in the literal *symmetric(X)*.

```
    assert(symmetric(f)),
    assert(symmetric(g)),
    assert(symmetric(h)),
    retract(symmetric(f)),
    symmetric(X),
```

retractall/1 (i)

This builtin predicate, standard to Prolog-like languages, is an extended version of retract/1. It takes a Prova list (which is the same as a compound term in Prova) and removes all facts in the current rulebase that are found using unification to be the same or a specialization of the supplied term.

Consider this example.

```
:- solve(test008(X,Y)).

test008(X,Y) :-
    assert(dual(h,a)),
    assert(dual(h,_)),
    asserta(dual(h,f)),
    retractall(dual(_,a)),
    dual(X,Y),
    println(["Rule B: dual",X,Y]," ").
```

It prints:

```
Rule B: dual h f
X=h, Y=f
Rule B: dual h <46>
X=h, Y=<46>
```

As we see, *retractall(dual(_*,a)) removes only the fact *dual(h,a)* but not *dual(h,_)*.

sendMsg/5 (io,i,i,i,i)

This built-in is used for sending the specified message immediately (contrast this with sendMsgSync). The predicate takes the following arguments.

1. *XID* - conversation id of the message;
2. *Protocol* - name of the message passing protocol;
3. *Destination* - a logical endpoint;
4. *Performative* - the message type broadly characterising the meaning of the message;
5. *Payload* - a Prova list containing the actual content of the message.

The following code responds to an inbound *init* message by sending two *request* messages to the logical endpoint "Agent002" via the ESB protocol.

```
rcvMsg(XID,Protocol,From,init,[]) :-
    agent001().

card('Mastercard').
card('Visa').

agent001() :-
    card(Card),
    sendMsg(XID,esb,"Agent002",request,card(Card)).
```

sendMsgSync/5 (io,i,i,i,i)

This built-in is used for sending the specified message **after** the current goal has been fully processed. The predicate takes the same arguments as sendMsg.

1. *XID* - conversation id of the message;
2. *Protocol* - name of the message passing protocol;
3. *Destination* - a logical endpoint;

4. *Performative* - the message type broadly characterising the meaning of the message;
5. *Payload* - a Prova list containing the actual content of the message.

The *sendMsgSync* predicate is different from sendMsg that sends the message **immediately** to the specified destination. Consider the following code:

```
switch_thread() :-
   % INCORRECT: must use sendMsgSync
   sendMsg(XID,task,0,switch,[]),
   rcvMsg(XID,task,From,switch,[]).
client() :-
   % Send all the test messages from a separate thread
   switch_thread(),

   % Use user-id as conversation-id (XID) for partitioning so that each user is processed sequentially
   sendMsg(user1,async,0,request,login(user1,'10.10.10.10')),
```

The idea here is to "jump threads" so that message sending occurs on an automatically selected thread from the **task** thread pool (see Concurrent reactive messaging). What actually happens is that once the *switch* message is sent, the following *rcvMsg* inside *switch_thread* makes sure there is an inline reaction waiting for a reply on the **task** protocol. So while *rcvMsg* **itself** is executed on the main thread, the reaction it creates should be ready to intercept the expected message on another **task** thread. We have a race condition, the reaction may not be ready before the message arrives.

The built-in *sendMsgSync* ensures that the current rule runs exhaustively until all solutions or failure is encountered, which means, among other things, that all subsequent *rcvMsg* are all executed, before the message is actually sent. So the correct code will be as above but with *sendMsgSync*.

```
switch_thread() :-
   % CORRECT: using sendMsgSync
   sendMsgSync(XID,task,0,switch,[]),
   rcvMsg(XID,task,From,switch,[]).
```

This message sending primitive must typically be used whenever there are send+receive pairs in the code and there is a possibility that the execution thread will be changed. The protocol **async** is specially designed for maintaining coherence through the whole message exchange in a conversation, in which case, a conversation is always mapped to one thread. The same is true for the default **self** protocol that runs everything in a single thread.

spawn/4 (io,i,i,i)

This built-in is used for running a Java method in a separate thread. It asynchronously returns the results in a separate message sent on the *self* protocol, i.e., received on the agent's main thread. The response is correlated using the conversation-id that is instantiated by the predicate if a free variable is supplied in the first argument, or on the specified conversation-id, if it is already set. Otherwise, the predicate accepts the following parameters.

1. *XID* - conversation id of the message with results;
2. *Target* - a Java object instance or a qualified Java class name;
3. *Method* - the method to be called;
4. *Args* - a Prova list with parameters to the call.

```
spawn(XID1,Object,Method,Args),
rcvMsg(XID1,Protocol,Me,return,Ret)
```

tokenize_enum/2 ([i,i],io) ND

This non-deterministic predicate enumerates delimited tokens in the input string. It is structured as a multi-valued Prova function (see Functional programming extensions). In the first argument, it accepts a Prova list with two input parameters: an input string and a delimiter string. The predicate produces independent solutions, one each for each possible token in the second argument. This creates a choice point with as many non-deterministic branches as there are tokens in the input stream.

The following example shows

```
:- solve(test_tokenize_enum_1(Value1)).

:- solve(test_tokenize_enum_2(Value2)).

test_tokenize_enum_1(Value) :-
   % non-deterministically enumerate delimited tokens in a string
   tokenize_enum(["j\t12s\ttt","\t"],Value).

test_tokenize_enum_2(tt) :-
   % non-deterministically enumerate delimited tokens in a string
   tokenize_enum(["j\t12s\ttt","\t"],tt).
```

This returns:

```
Value1=j
Value1=12s
Value1=tt
Value2=tt
```

tokenize_list/2 ([i,i],io)

This predicate tokenizes an input string using a supplied separator and returns a Prova list with tokens. The predicate is structured as a Prova function (see Functional programming extensions). In the first argument, it accepts a Prova list with two input parameters: an input string and a separator. The second argument is the produced list of tokens. If this list is supplied as input, it is unified against the output list tokens.

The following example demonstrates.

```
:- solve(test_tokenize_list("a,b,c",L)).

test_tokenize_list(In,[T|Ts]) :-
   % Create a list of tokens separated by ","
   tokenize_list([In,","],[T|Ts]).
test_tokenize_list(In,L) :-
   % Create a list of tokens separated by ","
   tokenize_list([In,","],L).
test_tokenize_list(In,ttt) :-
   tokenize_list(In,",",ttt).
```

This test prints the following solutions.

```
L=[a,b,c]
L=[a,b,c]
```

type/2 (i,io)

This predicate returns in the second argument the name of the Java class of the object supplied as input in the first argument. The output may already be set to a String value, which will then return success if the class name of the supplied object matches that string.

The following fragment will print the name of the *ArrayList* class..

```
   List=java.util.ArrayList(),
   type(List,TypeList),
   println([TypeList]),
```

This prints

```
java.util.ArrayList
```

unescape/2 (i,o)

This predicate takes a string with encoded escape characters and returns a string with the actual embedded control characters.

```
:- solve(p(Esc1)).

:- solve(q(Esc1)).

% One solution
p(Esc1) :-
   unescape("line1\nline2\nline3",Esc1).

% One solution
q(Esc1) :-
   unescape("line1\tline2\tline3",Esc1).
```

The above code prints.

```
Esc1=line1
line2
line3
Esc1=line1     line2   line3
```

unique_id/1 (o)

This built-in predicate returns a simple String that combines the logical agent name with the sequential *Long* value representing a unique occurrence inside the agent instance. It can be used for a variety of purposes, including generating unique partition ID's used by the partitioned reaction predicate *rcvMsgP*.

```
unique_id(PID1)
```

This would assign *PID1* to something like:

```
myagent:12345
```

2. Reactive messaging

2.1. Design principles

Reactive messaging in Prova is a fundamental part of the system that is used for organising distributed Prova engines into a network of communicating agents. It is also a foundation of the rule-based workflow and event processing functionality. The roots of the Prova reactive messaging capabilities lie in the series of papers written on so called Vivid Agents by Michael Schroeder and Gerd Wagner. A ***Prova agent*** is an instance of a running rulebase that includes message passing primitives.

Prova agents communicate using what we call ***protocols***. Inside a Prova agent rulebase, a protocol is just an argument passed to message passing primitives, both for receiving and sending messages, and a decoration attached to the actual messages. The distribution aspect of Prova agents is consequently not part of the Prova agent rulebases code at all, with the remote protocol, called *esb*, being one of those protocols that is provided by the Java container running the Prova engine.

The key principle used for message processing is pattern matching. This type of message handling is quite universally known and used in such communication languages as Erlang. The Prova pattern matching is similar but extends the Erlang type of pattern matching in a number of ways.

Prova message passing primitives use other standard, position-based, metadata attributes, in addition to the message protocol. These are ordered parameters to both message sending and receiving primitives:

1. ***XID*** - conversation id of the message;
2. ***Protocol*** - name of the message passing protocol;
3. ***Destination*** (on sending) or ***Sender*** (on receiving);
4. ***Performative*** - the message type broadly characterising the meaning of the message;
5. ***Payload*** - a Prova list containing the actual content of the message.

All these parameters can be used for constraining the reactions on the receiving side, thereby simplifying the coding of inter-agent communication. You specify all those parameters you want to be fixed as constants and leave those that require further reasoning about as free variables that will be assigned to when the actual message arrives. Sending messages reverses this process, so that you include the same above parameters as part of the message sending builtins, ***sendMsg*** or ***sendMsgSync***.

As described in Using reaction rules for receiving messages, there are two types of reaction rules: *global*, that are active for the whole lifetime of a rulebase, and *inline*, which are far more dynamic and whose scope can be controlled in a very flexible way, for example, by disabling and re-enabling them back again based on other reactions. Annotations for message receiving shows how to set a timeout on an inline reaction.

Event processing in Prova is based on collective ***group*** reactions. Such reactions do not work alone, instead many reactions are instantiated and respond to incoming messages collectively, for example, watching for a particular ***event pattern*** in the inbound message stream(s).

2.2. Sending messages

Message sending is an integral part of the Prova design. It is an operation that is a dual to message receiving (see Using reaction rules for receiving messages). Message sending is initiated when a Prova engine processes a literal in a rule body that is either of the two special builtins: ***sendMsg*** or ***sendMsgSync***. The duality with message receiving is clearly visible in the following structure of mandatory position-based arguments to these primitives.

2.2.1. Position-based arguments

XID - conversation id of the message;
Protocol - name of the message passing protocol;

Destination - a logical endpoint;
Performative - the message type broadly characterising the meaning of the message;
Payload - a Prova list containing the actual content of the message.

Here are some examples of message sending:

```
    sendMsg(XID,task,0,inter,[1])
    sendMsg(q1,async,0,request,login(user2,'30.30.30.30'))
    sendMsg(XID3,self,Me,queryref,[id0,parent(X,Y)])
    sendMsgSync(XID,esb,"Worker",request,c())
```

Conversation id XID

Broadly speaking, conversation id is used for message correlation. However, the accent here goes on the word "conversation", rather than "correlation". It is because the conversation identity is supposed to be preserved over relatively large, logically distinct ***conversations*** that typically achieve some goals in the participating agents. This is in contrast with correlation which simply guarantees the match between the request and response.

If this parameter is a free variable at the time ***sendMsg*** (or ***sendMsgSync***) is executed, it is set by the engine to a unique value representing a brand new conversation id and as a result this variable is bound to this value, which can be used immediately for further messaging. If the parameter is a constant value, the sent message is supposed to be a follow-up for an ongoing conversation (so that the particular conversation-id had been obtained from an earlier received message). It may also be the case that the sender agent is confident that the constant it chooses will be unique, which most often happens for demonstration or test examples.

Here is an example of a typical follow-up pattern:

```
branch(XID) :-
    @and(g1)
    rcvMsg(XID,Protocol,From,and,Events),
    sendMsg(XID,esb,"vm://global",job_completed,[]).
```

The above rule expects a message using ***rcvMsg*** and then follows on the same ***XID*** with a message sent using ***sendMsg*** to another destination on the specified protocol ***esb***.

Protocol

The message exchange protocol is currently taken from a hard-coded set of protocols that are built into Prova.

There are three **internal** protocols: *self*, *task*, *async*, and *swing*. The *swing* protocol cannot be currently used for message sending, only for receiving notifications from Swing components (see Reactive messaging for Java Swing). The other three protocols are discussed in Concurrent reactive messaging. In the main, if you want to ensure fully sequential processing of received messages, you might want to use the *self* protocol. The *task* protocol is an execution thread pool that makes no guarantees on the order in which the received messages will be processed, so that two messages for the same conversation-id might end up executing concurrently or even in a reverse order. This pool is used for running tasks achieving maximum throughput. The *async* protocol is quite different from the typical Actor languages in that it makes good use of the available conversation id to pin the processing to a unique thread as calculated from the conversation id. This is the preferred way to handle long-going conversations, to achieve correctness, maximise data locality, and minimise the need for synchronisation.

The *esb* protocol (named after ESB, Enterprise Service Bus) is used for designating the containing agent as the forwarder for dispatching the message. In the mule-prova-agents project, the Mule ESB anbled components are used as a container for Prova agents so that with the ***Protocol*** set to *esb* and ***Destination*** set to the logical endpoint on the Mule ESB, the message is delivered by Mule over any of its dozens of actual message transports to the agent that may be located locally or anywhere in the world.

In the Mule Pova agents project, when a Mule components ***receives*** a message from elsewhere, we swap the inbound protocol name from *esb* to *async* in order to guarantee that the conversations are processed on the *async* protocol. The Prova agent then is assumed to understand that although it receives messages on a nominally internal protocol *async*, in actual fact, it must use the *esb* protocol for responding (see ESB example).

Destination

For internal intra-agent messaging, the destination must be either a constant 0 or the actual name of the running agent. The latter is obtainable by using the builtin ***iam***:

```
test() :-
   iam(Me),
   % Send a queryref message with replies to be sent back to the agent
   sendMsg(XID,self,Me,queryref,parent(X,Y),id0).
```

The variable ***Me*** above is instantiated by ***iam*** to be the name of the current agent.

In the case of inter-agent distributed message sending, ***Destination*** is a logical name of the destination endpoint. In particular, if Mule ESB is used for message routing, it is a logical name of the endpoint as configured via the Mule configuration file. For example:

```
   <jms:endpoint name="Worker1" topic="worker" />
```

This isolates Prova code from unnecessary details related to the use message transports with Mule perfectly able to route the messages as required, including the use of interceptors or specialised routing, including sending the message to more than one actual destination.

Performative

Performatives (that we routinely call, ***message types***) is a concept derived from M. Searle's Speech Act theory and used in the FIPA Agent Control Language (see for example, http://jade.tilab.com/papers/AIIA-jvp-fb.pdf) for specifying widely recognised message types that agents can reason upon with actually inspecting the message payload. In FIPA ACL, they resemble to formalised typical questions and answers humans would use in normal conversation, for example, **ask-if** or **inform**.

Prova does not impose any particular discipline on using performatives but allows agent authors to easily encode performative based reasoning. A number of examples included in the Prova distribution make use of performatives to great effect. This is taken from **reloaded/test023.prova**:

```
test023() :-
   ...
   % Send a queryref message with replies to be sent back to the agent
   sendMsg(XID3,self,0,queryref,parent(X,Y)),
   rcvMult(XID3,self,Me,reply,parent(X,Y)),
   println(parent(X,Y)),
```

Processing messages with the **queryref** performative typically involve local derivation of the query included as the message payload and sending back of the answerset facts as individual messages:

```
% Reaction rule to general queryref
rcvMsg(XID,Protocol,From,queryref,[X|Xs]) :-
   derive([X|Xs]),
   % As many messages as results are sent here
   sendMsg(XID,Protocol,From,reply,[X|Xs]).
```

Payload

Payload is the main information content of a message (or event). In Prova, the content **must be** a ***ProvaList***, which, obviously, is a container that can hold other data. It must be noted, however, that each element in a ***ProvaList*** must be derived from a class ***ProvaObject***, which includes Prova constants, variables, or other lists. If the message is sent from a Prova agent, all of this is satisfied automatically.

```
    sendMsg(user2,async,0,request,login(user2,'30.30.30.30'))
```

In this example, the payload ***login(user2,'30.30.30.30')*** automatically becomes a Prova list (note that in Prova, a(b,c) is the same as [a,b,c]).

If a pure Java code is used for sending messages to a Prova agent via ***ProvaCommunicator***, you can use the following method to wrap the Java ***Object[]*** array automatically.

```
ProvaCommunicator.sendMsg(String xid, String protocol, Object obj_receiver, String perf, String term,
Object[] objs)
```

Alternatively, the following example shows how to use a low-level ***ProvaCommunicator.addMsg()*** method to send a Prova message to a Prova agent from Java:

```
    ProvaList terms = ProvaListImpl.create( new ProvaObject[] {
        new ProvaConstantImpl(queryId),
        new ProvaConstantImpl("async"),
        new ProvaConstantImpl(0),
        new ProvaConstantImpl("event"),
        ProvaListImpl.create(new ProvaObject[] {
            new ProvaConstantImpl(msg)
            })
    });

    comm.addMsg(terms);
```

Now going back to the ***Payload*** parameter, the fifth argument to the ProvaList factory method above is the payload containing a ***msg*** object with the actual data. Note that this object is wrapped by a ***ProvaConstant***.

One sensible approach is to use a Java ***Map*** for the payload ***msg***. The following example shows how the data can be extracted from this ***Map*** in a reaction guard so that the reaction can pattern match the inbound message.

```
pattern('groupby_rate',QID,Properties) :-
    Timer = Properties.get("timer"),
    Field = Properties.get("field"),

    Counter = ws.prova.eventing.MapCounter(),
    @group(g1) @timer(Timer,Timer,Counter)
    rcvMsg(QID,async,From,event,[Msg]) [Group=Msg.getObject(Field),Counter.incrementAt(Group)].
```

Alternatively, if the message payload is passed as a proper ***ProvaList***, the reaction ***rcvMsg*** can pattern match the payload ***login(User,IP2)*** directly against the pattern including local variables, constants or embedded lists as well as use reaction guards if necessary:

```
server_1(XID,User,IP) :-
    @group(g1)
    rcvMsg(XID,Protocol,From,request,login(User,IP2)) [IP2!=IP],
    println(["Suspicious login",User,IP,IP2]," ").
```

2.2.2. Synchronized message sending with sendMsgSync

As the primitive ***sendMsg*** is executed on the current thread, the message is sent immediately to the specified destination. Consider the following code:

```
switch_thread() :-
   % INCORRECT: must use sendMsgSync
   sendMsg(XID,task,0,switch,[]),
   rcvMsg(XID,task,From,switch,[]).
client() :-
   % Send all the test messages from a separate thread
   switch_thread(),

   % Use user-id as conversation-id (XID) for partitioning so that each user is processed sequentially
   sendMsg(user1,async,0,request,login(user1,'10.10.10.10')),
```

The idea here is to "jump threads" so that message sending occurs on an automatically selected thread from the **task** thread pool (see Concurrent reactive messaging). What actually happens is that once the ***switch*** message is sent, the following ***rcvMsg*** inside ***switch_thread*** makes sure there is an inline reaction waiting for a reply on the **task** protocol. So while ***rcvMsg*** **itself** is executed on the main thread, the reaction it creates should be ready to intercept the expected message on another **task** thread. We have a race condition, the reaction may not be ready before the message arrives.

The primitive ***sendMsgSync*** ensures that the current rule runs exhaustively until all solutions or failure is encountered, which means, among other things, that all subsequent ***rcvMsg*** are all executed, before the message is actually sent. So the correct code will be as above but with ***sendMsgSync***.

```
switch_thread() :-
   % CORRECT: using sendMsgSync
   sendMsgSync(XID,task,0,switch,[]),
   rcvMsg(XID,task,From,switch,[]).
```

This message sending primitive must typically be used whenever there are send+receive pairs in the code and there is a possibility that the execution thread will be changed. The protocol **async** is specially designed for maintaining coherence through the whole message exchange in a conversation, in which case, a conversation is always mapped to one thread. The same is true for the default **self** protocol that runs everything in a single thread.

2.3. Using reaction rules for receiving messages

Prova rules are Horn rules with the head literal and a number of body literals. Reactive Messaging in Prova does change this pattern but allows these literals to have special meaning if they are message passing primitives.

2.3.1. Global reaction rules

The simplest form of a reactive rule is one when the **head** of the rule is a message receiving primitive distinguished by the ***rcvMsg*** predicate symbol. Here is an example of such rule:

```
rcvMsg(XID,Protocol,From,request,Proposal) :-
   println(["Received: ",Proposal]),
   process1(From,Proposal,Offer).
```

This rule has a rulebase lifetime scope, i.e., it is active while the rulebase runs in a Prova engine. This global scoping means that the rule is ready to receive any number of messages as they arrive to the agent. We discuss the concurrency issues in Concurrent reactive messaging.

The intended meaning of this construct is to wait messages that match the pattern specified in brackets after ***rcvMsg*** and respond to such matching messages with logic reasoning or side-effect causing primitives contained in the body of the rule, in this example, printing the message payload ***Proposal*** and processing it in order to generate an ***Offer*** (the example does not show what happens with the said ***Offer***).

Note that the message metadata, including conversation-id ***XID***, ***Protocol***, and the sender ***From***, have been left as free variables in the ***rcvMsg*** pattern, so that, for example, messages from any senders will be accepted.

2.3.2. Inline reaction rules

These reaction rules are more dynamic and volatile. Their scope can be controlled in a variety of ways, including restricting them to accept just one message, a specified number of messages, or be limited by a timeout. This type of reaction rules is especially useful for workflow and event processing.

The fundamental idea behind inline reaction rules is comparable to closures or continuations that are becoming more and more acceptable even in procedural languages. The reaction is created as part of evaluating the body of a rule when a message receiving primitive ***rcvMsg*** is part of that body.

Consider the following example:

```
branch(XID) :-
   spawn(XID,java.lang.Thread,sleep,1000L),
   rcvMsg(XID,self,Me,return,Ret),
   best(XID,BestService,BestOffer),
   println(["Received best offer: ",BestOffer," from ",BestService]).
```

What we do here is the following: the ***spawn*** primitive initiates a static Java method call to sleep for one second. This sleep occurs in a separate thread in a special ***task*** thread pool. Once the task is spawned, the ***rcvMsg*** is evaluated. This results in the creation of a closure that contains all the remaining literals in the body of the rule along with a dynamically generated reaction that is waiting for the pattern specified in brackets after ***rcvMsg***. This pattern indicates an interest in a message of pre-defined type ***return*** on the internal ***self*** protocol. This reaction will fire only once, when the matching message corresponding to a return from a spawned task is received. Note that the conversation-id ***XID*** is used for correlation.

The takeaway from this is that ***rcvMsg*** inside the body of any Prova rule results in a creation of a temporal reaction that freezes the current state of all the context and body literals following this ***rcvMsg***. The temporal reaction does not consume any thread and is just a pure stored data that gets matched against by the Prova agent each time it perceives a new inbound message on the matching protocol. Once the matching message is received, that stored data is fully destroyed.

2.3.3. Controlling the message reaction multiplicity

The ***rcvMsg*** primitive on its own "works" only once. If we want to keep the reaction alive indefinitely, the simplest way to achieve that is to replace it with the ***rcvMult*** primitive.

```
server() :-
   % Start detection on each new login
   rcvMult(XID,Protocol,From,request,login(User,IP)),
   server_1(XID,User,IP).
```

This construct is, for example, useful in event processing applications for specifying a pattern initiator message. The example above waits indefinitely for logins passing the relevant data to further processing in rules for predicate symbol ***server_1***. You may ask a question, why could not we have used a global reaction rule for this prupose and the answer will be yes, we could have, in this particuar instance. What about the case when a multiple reaction is part of the response to another previous inline reaction? This makes the reaction indefinite but specific to the context of the preceding reaction.

The example below shows another way to specify multiple reactions and shows how such reactions can be part of a follow-up for a previous reaction. Let us just add one rule for ***server_1*** from the previous example.

```
server_1(XID,User,IP) :-
```

```
% Indefinitely receive purchase events that follow the previous login
@group(g1) @count(-1)
rcvMsg(XID,Protocol,From,request,purchase(User,IP2)) [IP2!=IP],
println(["Suspicious purchase",User,IP,IP2]," ").
```

We use an annotation *@count* with parameter *-1* to indicate that the inline ***rcvMsg*** reaction will be indefinite. This only works in event processing groups (hence the ise of the *@group* annotation). The variables ***XID***, ***User*** and ***IP*** are already concrete data since we are in the context of the previous reaction to ***login***. The guard in square brackets (see Guards in message processing) allows us to specify a constraint on the payload. Guards are executed **before** the message is accepted by the reaction. This means that if the guard fails, the reaction stays intact even if it is a one-message ***rcvMsg*** reaction. Any logical Prova code can go into the guard, not just simple arithmetic constraints, which makes it a very powerful feature.

2.3.4. Explicit termination of inline reactions

To explicitly terminate an inline, particularly multiple, reaction one has to send a specially formatted message to the reaction. The test ***msg006.prova*** shows how reaction termination works. Note that that the "Received" is printed from a conversation thread selected based on conversation-id so its order with respect to "Sent"'s is random. However, the termination signal is guaranteed to arrive before the third send is receved by the target conversation thread, so the third reaction never happens.

```
    % This example demonstrates termination of a multiple inline reaction.
    % It works by an agent instructing the receiving reaction matching a template to terminate (eof).
    % Message ordering within the same conversation is ensured by conversation-id XID always mapped to
the same thread
    % from the thread pool based on partitions.

    :- eval(msg006()).

    msg006() :-
        println(["==========Messaging test 006=========="]),

        % This reaction will stay active after receiving the first message until it is terminated by the eof
message
        rcvMult(XID,Protocol,From,inform,a(I)),
        println(["Received: ",rcvMult(XID,async,From,inform,a(I))],"").
    msg006() :-
        % Use the 'async' verb to force the reactions to run on the thread pool ('self' can be used to run on the
main agent thread)
        sendMsg(XID,async,0,inform,a(1)),
        println(["Sent: ",sendMsg(XID,async,0,inform,a(1))],""),
        % Send termination signal
        sendMsg(XID,async,0,eof,[ReactionXID,Protocol,From,inform,a(_)]),
        println(["Sent: ",sendMsg(XID,async,0,eof,[ReactionXID,Protocol,From,inform,a(_)])],""),
        % The next message will be ignored as rcvMult will have terminated
        sendMsg(XID,async,0,inform,a(2)),
        println(["Sent: ",sendMsg(XID,async,0,inform,a(2))],"").
```

This example prints:

```
Sent: [sendMsg,prova:1,async,0,inform,[a,1]]
Received: [rcvMult,prova:1,async,0,inform,[a,1]]
Sent: [sendMsg,prova:1,async,0,eof,[<X28>,<X29>,inform,[a,<X31>]]]
Sent: [sendMsg,prova:1,async,0,inform,[a,2]]
```

2.4. Guards in message processing

The guards mechanism in reactive messaging uses the general guard construct available in the Prova language (see Guards in Prova). We discuss here only the use of guards for individual, not Event Processing groups based, reactions.

When a Prova agent perceives an inbound message, a pattern match is attempted for all five position-based parts of the message, including the message payload. We are particularly interested in the case when the match is made with an inline reaction ***rcvMsg***. Since ***rcvMsg*** can only accept one message, without the guards extension, matching the pattern would have immediately invalidated the reaction for any further matching, even if a condition that was following the reaction failed. Consider the following two examples:

post condition

```
IP='10.10.10.10',
rcvMsg(XID,Protocol,From,request,login(User,IP2)),
IP2!=IP,
println([IP2]).
```

pre condition (guard)

```
IP='10.10.10.10',
rcvMsg(XID,Protocol,From,request,login(User,IP2)) [IP2!=IP],
println([IP2]).
```

In the first case, even if the received message contained exactly the same ***IP2*** as ***IP***, the fact that this invalidates the condition would be detected too late, and finding another correctly qualifying inbound message would be impossible as the reaction would be satisfied and gone. In the second case, the guard is tested right after pattern matching but **before** the message is fully accepted, so that the net effect of the guard is really to serve as an extension of pattern matching for literals.

Another great use of guards is for extracting field values from the message payload, see Extracting payload fields using guards.

2.5. Using slotted terms (Prova maps) in reactions

Prova 3.0 messaging can use Prova maps as part (or the whole) of their payload. The patterns in inline reactions and reaction rules can include Prova maps for matching against the incoming messages (events) directly during the unification process. Importantly, the patterns may include only a subset of the fields in the actual message with the understanding that only the fields included in the pattern must match the fields in the message. The same rule applies to Java-typed variables in the patterns. Objects inside the incoming messages must be of the same type or a subtype of the type specified in the pattern.

The following example ***msg011.prova*** demonstrates this usage idiom. Note that the pattern in ***rcvMsg*** only includes the field ***b***. This allows the (closure) code that follows the reaction to only have access to that field. However, importantly, the message that has the matching field ***b*** is accepted regardless of any other fields it might have (in this instance, ***a***). Finally, the use of type prefix in ***Integer.I*** could have been omitted, as the inbound messages indeed have an integer value for the field ***b***.

```
% Send and receive a message containing a Prova map payload.

:- eval(msg011()).

msg011() :-
   println(["==========Messaging test 011=========="]),

      % Only one message will be received here as the reaction disappears after the first message is
consumed
```

```
        % Messaging is contra-variant in that the more general pattern in the reaction subsumes the message
parameters
        rcvMsg(XID,Protocol,From,inform,{b->Integer.I}),
        println(["Received: ",rcvMsg(XID,self,From,inform,{b->Integer.I}]]).
    msg011() :-
        for([1,2],I),
        sendMsg(XID,self,0,inform,{a->1,b->I}),
        println(["Sent: ",sendMsg(XID,self,0,inform,{a->1,b->I})],"").

    % This outputs:
    % ==========Messaging test 011==========
    % Sent: [sendMsg,prova:1,self,0,inform,{b=1, a=1}]
    % Sent: [sendMsg,prova:2,self,0,inform,{b=2, a=1}]
    % Received: [rcvMsg,prova:1,self,0,inform,{b=1}]
```

2.6. Concurrent reactive messaging

Prova agents execute protocols and send and receive messages asynchronously. The ***rcvMsg*** and ***rcvMult*** builtins do not block the processing thread but instead immediately release it, preserving all the current rule context so that when a matching message arrives, the processing resumes as though it had never been interrupted. Now the big question is, on what thread the processing is resumed?

Prova engine runs on the main thread and two additional thread pools: a ***conversation*** thread pool and a ***task*** thread pool. All common goals are run on the main thread but the thread pool a message reaction initiated by ***rcvMsg/rcvMult*** runs on is chosen by the protocol value passed to an inline reaction ***rcvMsg/rcvMult*** that receives a message initiating the goal processing (remember message reactions are goals): the *self* protocol now targets specifically the main thread, the *async* protocol targets the ***conversation*** pool, and the *task* protocol targets the ***task*** pool. The difference between the two thread pools is that in the *async* case, the particular thread from the conversation pool is **uniquely** chosen based on the conversation-id ***XID*** passed with the inbound message. This means that messages belonging to the same conversation are always processed on the same thread. The added benefit of this is that any local data in context of the rule containing ***rcvMsg*** are processed only on one thread which reduces context switching on multi-core architectures. The choice of the thread in the task pool is completely random.

Typically, an adaptor implemented in Java adds inbound messages into the Prova engine by using the *async* protocol. This ensures that the messages for the same conversation-id are processed by the agent on the same thread. This is a specific example of the Mule ESB adaptor that sets the second parameter of the message to "async" and adds the message to the ProvaCommunicator queue.

```
    /**
     * Process an inbound Prova message that is assumed to arrive on this endpoint
     */
    public Object onCall(MuleEventContext context) throws Exception {
         // Extract Prova RMessage
         MuleMessage inbound = context.getMessage();
         ProvaList terms = null;
         if( inbound.getPayload() instanceof ObjectMessage )  {
           terms = (ProvaList) ((ObjectMessage) inbound.getPayload()).getObject();
         } else {
           terms = (ProvaList) context.getMessage().getPayload();
         }

         // Add the message as a goal to the asynchronous Prova Communicator queue
           terms.getFixed()[1] = new ProvaConstantImpl("async");
       comm.addMsg(terms);
       // We are done, everything is asynchronous
       context.setStopFurtherProcessing(true);

         return null;
    }
```

The following extensive example illustrates how *async* and *task* may interoperate in an interesting way. A general assumption is that the reactions belonging to the same conversation run sequentially. However, imagine that while processing a reaction to a message on conversation ***XID***, a rule sends more messages and uses ***rcvMsg*** to accept the response. Clearly, if no special care is taken, if another message arrives on the original conversation, it may be processed earlier than the mentioned response arrives, which may in some cases be undesirable. The test **msg010.prova** shows a particular message-based locking mechanism ensuring that the original messages are processed sequentially. The code uses a ***Requestor*** and a ***Lock Manager*** to make the messages going via ***Requestor*** execute sequentially. Note that since for any ***XID***, the locks are processed on the same thread, they are never in contention and no synchronisation is required. The code of ***msg010.prova*** follows.

```
% Concurrent Prova messaging
% Demonstrate the use of 'cycled' event processing.
% Three inbound events for each of two different XID are executed concurrently.
% The events for the same XID must be processed sequentially so that a new event is executed
%           on the same 'conversation' thread when the previous processing is done.
% Internally, each processing step consists of several concurrent processes executed on the 'task' thread pool.
% Only when all of them are finished, the lock manager unlocks the processing of further events on the same XID.

%%%%%%%%%%%%%%%%%
%%% REQUESTOR %%%
%%%%%%%%%%%%%%%%%

rcvMsg(XID,async,From,raw,Payload) :-
    sendMsg(XID,async,0,lock,[]),
    rcvMsg(XID,async,From,locked,[]),
    !,
    process(XID,Payload).

%%%%%%%%%%%%%%%%%%
%% LOCK MANAGER %%
%%%%%%%%%%%%%%%%%%

:- eval(lock_manager_init()).

lock_manager_init() :-
    $PLock=java.util.concurrent.ConcurrentHashMap(),
    $PJoin=java.util.concurrent.ConcurrentHashMap().

lock_manager_check(XID) :-
    Lock=$PLock.get(XID),
    Lock>0,
    !,
    fail().
lock_manager_check(XID) :-
    $PLock.put(XID,1).

rcvMsg(XID,async,From,lock,[]) :-
```

```
        lock_manager_check(XID),
        sendMsg(XID,async,0,locked,[]).
    rcvMsg(XID,async,From,unlock,[]) :-
        $PLock.put(XID,0),
        println([unlocked,XID],": "),
        sendMsg(XID,async,0,locked,[]).

    partition_join_init(XID) :-
        $PJoin.put(XID,0).

    partition_join_increment(XID,N2) :-
        N=$PJoin.get(XID),
        N2=N+1,
        $PJoin.put(XID,N2).

    %%%%%%%%%%%%%%%%%%%
    %%%%% SERVER %%%%%
    %%%%%%%%%%%%%%%%%%%

    :- eval(server()).

    % All requests run in parallel threads from the 'task' thread pool.
    % This happens due to the client specifying 'task' as the protocol (second parameter to sendMsg).
    server() :-
        println(["==========msg010=========="]),
        rcvMult(XID,Protocol,From,run,t(A,B,I),I),
        I2=I*2,
        sendMsg(XID,async,From,result,t(I2),I).

    %%%%%%%%%%%%%%%%%%%
    %%%%% CLIENT %%%%%
    %%%%%%%%%%%%%%%%%%%

    :- eval(client()).

    client() :-
        element(I,[1,2]),
        % For each I, the conversation-id XID is different
        % Initialise XID so that the subsequent responses are all executed on the same thread
        sendMsg(XID,task,0,noop,[]),
        element(J,[1,2,3]),
        sendMsg(XID,async,0,raw,[I,J]),
        rcvMsg(XID,async,0,raw_result,Result,J),
        println(["Result:",I,J,Result]," ").

    %%%%%%%%%%%%%%%%%
    %%% PROCESSOR %%%
    %%%%%%%%%%%%%%%%%

    process(XID,[I,J]) :-
        partition_join_init(XID),
        element(K,[1,2,3,4]),
        % Execute on the non-partitoned 'task' thread pool
        sendMsg(XID,task,0,run,t(I,J,K),K),
        % For each response from the server, the remainder of this rule's body
        %   runs in one and the same "conversation" thread chosen from the partitioned conversation thread
pool using XID as key.
        %   This means that there is no synchronisation required.
        rcvMsg(XID,async,From,result,t(K2),K),
        partition_join_increment(XID,N2),
        wrapup(XID,[I,J],N2).

    % This only gets executed when all responses have arrived
    wrapup(XID,[I,J],4) :-
        sendMsg(XID,async,0,raw_result,[4],J),
        sendMsg(XID,async,0,unlock,[]).
```

This is the output from the above example:

```
==========msg010==========
locked: prova:2
locked: prova:1
prova:2@2@1@1
prova:1@1@1@1
prova:1@1@1@2
2>1>1>1>2
1>1>1>1>2
prova:1@1@1@3
prova:2@2@1@2
prova:2@2@1@3
1>1>2>2>4
2>1>2>2>4
2>1>3>3>6
prova:2@2@1@4
1>1>3>3>6
2>1>4>4>8
done[2,1]4
Result: 2 1 [4]
unlocked: prova:2
locked: prova:2
prova:1@1@1@4
prova:2@2@2@1
prova:2@2@2@2
prova:2@2@2@3
prova:2@2@2@4
2>2>1>1>2
1>1>4>4>8
done[1,1]4
Result: 1 1 [4]
2>2>2>2>4
unlocked: prova:1
2>2>3>3>6
locked: prova:1
2>2>4>4>8
done[2,2]4
Result: 2 2 [4]
unlocked: prova:2
prova:1@1@2@2
prova:1@1@2@3
prova:1@1@2@1
prova:1@1@2@4
locked: prova:2
1>2>1>2>4
prova:2@2@3@1
prova:2@2@3@2
2>3>1>1>2
2>3>2>2>4
1>2>2>3>6
1>2>3>1>2
```

```
1>2>4>4>8
done[1,2]4
Result: 1 2 [4]
prova:2@2@3@3
2>3>3>3>6
prova:2@2@3@4
unlocked: prova:1
locked: prova:1
2>3>4>4>8
done[2,3]4
Result: 2 3 [4]
unlocked: prova:2
prova:1@1@3@2
prova:1@1@3@3
prova:1@1@3@4
1>3>1>2>4
1>3>2>3>6
1>3>3>4>8
prova:1@1@3@1
1>3>4>1>2
done[1,3]4
Result: 1 3 [4]
unlocked: prova:1
```

2.7. Comparison with Scala actors and delimited continuations

As described in http://lamp.epfl.ch/~rompf/continuations-icfp09.pdf in Section 4.4, Scala actor model suffers from the lack of composability due to ***react*** limited to the explicitly provided closure. In particular, this code below will not work.

```
def establishConnection() = {
   server ! SYN
   react {
      case SYN_ACK =>
      server ! ACK
   }
}

actor {
   establishConnection()
   transferData()
   ...
}
```

The paper proposes a solution using continuation based ***reset*** together with ***proceed*** to create a common context for composed behaviors.

```
def establishConnection() = {
   server ! SYN
   proceed(react) {
      case SYN_ACK =>
      server ! ACK
   }
}
actor {
   reset {
      establishConnection()
               transferData()
      ...
```

```
    }
}
```

In contrast, Prova allows direct compositions of protocol fragments.

```
establishConnection(XID) :-
    sendMsg(XID,esb,server,syn,[]),
    rcvMsg(XID,Protocol,server,syn_ack,[]),
    sendMsg(XID,esb,server,ack,[]).

transferData(XID,Payload) :-
    sendMsg(XID,esb,server,data,Payload).

actor(Payload) :-
    establishConnection(XID),
    transferData(XID,Payload).
```

2.8. Annotations for message receiving

We discuss here only the basic annotations that apply to both individual inline reactions and grouped reactions used for event processing. Extensive annotation support for event processing is discussed in Annotations for reaction groups.

@timeout

At this time, the only parameter this annotation can take is the timeout in milliseconds which starts at the moment the ***rcvMsg*** statement is executed (and consequently, an inline reaction is created). After the timeout elapses, the inline reaction is purged and is obviously no longer active.

```
server() :-
    % Start detection on each new login
    rcvMult(XID,Protocol,From,request,login(User,IP)),
    % Wait for a right follow-up while ignoring anything that does not match
    @timeout(1000)
    rcvMsg(XID,Protocol,From,request,login(User,IP2)) [IP2!=IP],
    % Once the full match has occurred, the above rcvMsg reaction is removed
    println([User,IP,IP2]," ").
```

2.9. Reactive messaging for Java Swing

As described in the Section on the built-in predicate *listen*, Prova rulebase can register a listener for *action*, *change*, *mouse* and *motion* Swing events. Behind the scenes, the class ProvaSwingAdaptor is the class that is the actual listener and all it does is mapping of the detected Swing events to messages that get executed on a specially designated Swing thread.

We reproduce below the messages this listener sends to the Prova rulebase on receiving events from UI. The messages are always sent on conversation-id fixed to the String "s".

```
% For action events:
sendMsg(s,task,0,swing,[action,ActionCommand,Source,ActionEvent])
% For state changed events
sendMsg(s,task,0,swing,[change,Source,ChangeEvent])
% For mouse click events
sendMsg(s,task,0,swing,[mouse,clicked,Source,MouseEvent])
% For mouse entered events
sendMsg(s,task,0,swing,[mouse,entered,Source,MouseEvent])
% For mouse exited events
sendMsg(s,task,0,swing,[mouse,exited,Source,MouseEvent])
% For mouse pressed events
sendMsg(s,task,0,swing,[mouse,pressed,Source,MouseEvent])
% For mouse released events
sendMsg(s,task,0,swing,[mouse,released,Source,MouseEvent])
```

All the rulebase has to do then is initialize reactions for message patterns matching the above messages. This is an example of a global reaction.

```
% Reaction to incoming swing action messages on buttons.
rcvMsg(XID,Protocol,From,swing,[action,Cmd,javax.swing.JButton.JB|Extra]) :-
   process_button(Cmd).
```

This is a detection of an event pattern corresponding to a mouse gesture using the Prova reaction groups.

```
detect_drag2(JB1,JB2) :-
   @group(g2)
   rcvMsg(s,Protocol,From,swing,[mouse,released,Src,Event])
[E1=javax.swing.SwingUtilities.convertMouseEvent(Src,Event,JB1),P1=E1.getPoint(),Boolean.TRUE=JB1.cont
ains(P1)].
detect_drag2(JB1,JB2) :-
   @group(g2) @not
   rcvMsg(s,Protocol,From,swing,[mouse,released,Src,Event]).
detect_drag2(JB1,JB2) :-
   @and(g2)
   rcvMsg(s,Protocol,From,and,Events),
   println(["Gesture detected"]).
```

We recommend that you study the included example *swing_rx.prova* that follows and extends the following example of the Rx framework that is part of .NET 4.0: http://themechanicalbride.blogspot.com/2009/07/developing-with-rx-part-1-extension.html. In particular, the "click extension event" is significantly more compact and efficient in Prova and is also extended in swing_rx.prova to a more complex case that involves negation.

3. Event processing

3.1. Event processing using reaction groups

Consider reactive functionality available in Prova (see Reactive messaging). If we wanted to "string" together a number of reactions in order to detect an event pattern, the simplest idea would be to use one reaction per event type that is part of the pattern. In fact, we could actually implement a "followed by" operator pretty easily, by literally following up one reaction with another as in the example below.

```
server_1(XID) :-
   rcvMsg(XID,Protocol,From,request,logout(User,IP)),
   println(["Got 1"]),

   rcvMsg(XID,Protocol,From,request,login(User,IP2)) [IP2!=IP],
   println(["Got 2"]).
```

This is brilliant, we could even set constraints on the received messages using guards.

There are many gaps in this first attempt. Let us consider how we could introduce a timeout.

```
server_1(XID) :-
   rcvMsg(XID,Protocol,From,request,logout(User,IP)),
   println(["Got 1"]),

   @timeout(1000)
   rcvMsg(XID,Protocol,From,request,login(User,IP2)) [IP2!=IP],
   println(["Got 2"]).
```

Note the use of the *@timeout* annotation above. This is an improvement as we are now able to wait for the follow-up event for a limited amount of time.

This is still imperfect, what if the first message fails to materialise at all? What we need is a collective reaction, a notion of a ***group*** context associated with a number of individual reactions. We want to associate a timeout with that group as a whole while still being able to specify timeouts on individual reactions. Enter the *@group* annotation.

```
server() :-
   % Start detection on each new login
   rcvMult(XID,Protocol,From,request,login(User,IP)),
   server_1(XID).

server_1(XID) :-
   @group(g1)
   rcvMsg(XID,Protocol,From,request,logout(User,IP)),
   println(["Got 1"]),

   @group(g1) @timeout(1000)
   rcvMsg(XID,Protocol,From,request,login(User,IP2)) [IP2!=IP],
   println(["Got 2"]).
server_1(XID) :-
   @and(g1) @timeout(2000)
   rcvMsg(XID,Protocol,From,and,Events),
   println(["Pattern detected: ",Events]," ").
```

The idea here is to group the reactions in one ***AND*** group and say "we want both of these events to arrive". The group members, that we call *event channels* are decorated with the *@group* annotation that can carry a list of logical group names. You may ask about the scope of this logical group name "g1" in this example. The group "g1" is just a template, the actual group instance is created each time the ***login*** event is perceived in the very first clause for predicate ***server***. All of those concrete group instances then co-exist and concurrently detect the follow-up sequences of a ***logout*** followed by a ***login*** from another ***IP***.

We limit the detections to a "followed by" pattern by allowing the second reaction to only occur after the first, inthe first clause for the group.

The all-important second clause for the group reaction incluses a ***group reaction*** annotated with the type of the group, which is *@and* in this instance. This is exactly where we attach additional constraints on the pattern reaction as a whole, so we place the second group-wide *@timeout* there.

What exactly is this *group reaction*? When the Prova engine finds a group, it builds all the group members and observes the semantics of the group. If it is an ***AND*** group, when all the conjuncted reactions are successfully detected (within the overall specified timeout), the special event of message type ***and*** is sent internally and its payload, captured in the variable ***Events***, contains the full history of events that resulted in the pattern detection.

To recap, event processing in Prova is based on reactive messaging augmented with the ability to group reactions together so that they "collaborate" for the event patterns detection.

3.1.1. White box or workflow event processing

It is critical to observe that each reaction in the pattern still retains its context. This style of reacting is different from typical stream processing engines because it leaves the workflow associated with the detection process open for arbitrary logical or procedural extensions whenever each individual event in the pattern is detected. For example, we can log or send messages right from within the pattern. This becomes particularly important when we want to achieve a measure of ***completeness*** associated with a pattern detection.

Consider a refinement process whereby a pattern becomes increasingly more complicated as we strive for detection precision. The adverse effect of this is the loss of generality, the pattern may become overly specialised and brittle. The workflow languages recognise the need for fault or compensation handlers and that is precisely what is lacking in stream processing. The collective Prova reaction style leaves the box open for workflow inspection and manipulation so that a pattern becomes much more accommodating to variation.

The second distinguishing feature of the event processing in Prova is that timeouts or partial detections result in the full trace of partial results being emitted in a special message with type ***timeout***.

```
server_1(XID,User,IP) :-
   @or(g1) @timeout(1000)
   rcvMsg(XID,Protocol,From,or,Events),
   println(["Pattern detected: ",Events," "]).
server_1(XID,User,IP) :-
   @or(g1)
   rcvMsg(XID,Protocol,From,timeout,Events),
   println(["Timeout occurred: ",Events," "]).
```

The example above contains an entirely optional second clause that reacts to this internal ***timeout*** event and, in this instance, simply prints these partial results. This could be used for diagnosing why the pattern was not fully detected and organising possible counter-measures.

These design decisions help to increase the robustness of event processing in Prova.

3.2. Structure of reaction groups

Event algebras are a formalism for describing ***composite events*** that are groups of other events satisfying specified algebraic constraints. Reaction groups is the mechanism that embeds event algebra into the Prova language.

Prova is well equipped for dealing with reactive behaviour (see Design principles) as it has basic constructs for receving messages that can be used as an event delivery mechanism. The ***rcvMsg*** and ***rcvMult*** primitives receive either single or multiple messages from within the body of the rules.

These primitives are non blocking, and instead store a continuation to the remaining trail of literals, which is activated once the matching message arrives. This message receiving capability is very much like that in Erlang, including the ability to use guards on reactions. However, Prova is quite different from Erlang. For example, Prova does not immediately consume the message but allows it to be accepted by many alternative reactions. There is a way to absorb a message by using the Prova (Prolog) CUT after the reaction.

Let us now talk more concretely about the event algebra in Prova. We need a way to group more than one reaction using a logical operator, for example, ***AND***, requiring that ***all*** events matching the reactions belonging to the group arrive (typically, within the specified period of time). This means we need to designate reactions as belonging to a group and a way to indicate somewhere the operator to be used. But there is more: the result of a composite event detection has to be represented somehow, so that it could be captured in the way the composed events are, as well as allow the resulting composite event to be used in other event groups with possibly different operators, creating a recursive operator expression.

An extended example below has all the main ingredients. We start with the code for the server that detects event patterns.

```
:- eval(server()).

server() :-
   % Start detection on each new login
   rcvMult(XID,Protocol,From,request,login(User,IP)),
   server_1(XID,User,IP).

server_1(XID,User,IP) :-
   @group(g1)
   rcvMsg(XID,Protocol,From,request,login(User,IP2)) [IP2!=IP].
server_1(XID,User,IP) :-
   @group(g1)
   rcvMsg(XID,Protocol,From,request,logout(User,IP)).
server_1(XID,User,IP) :-
   @and(g1) @timeout(2500) @group(g2)
   rcvMsg(XID,Protocol,From,and,Events),
   println(["AND detected: ",Events," "]).
server_1(XID,User,IP) :-
   @group(g2)
   rcvMsg(XID,Protocol,From,request,update(User,IP)).
server_1(XID,User,IP) :-
   @or(g2) @timeout(1000)
   rcvMsg(XID,Protocol,From,or,Events),
   println(["Pattern detected: ",Events," "]).
```

Once the **initiator event** (a user login) is detected by ***rcvMult*** in the rule for ***server***, the Prova code for the predicate ***server_1*** creates five reactions that simultaneously wait for subsequent events. For each new initiator event, more reactions will become active. However, as already mentioned, Prova does not block on any active reactions but instead keeps them in memory ready to match when qualifying inbound messages are detected. Now look closer at the annotations on those five reactions. The first two belong to the group ***g1***, indicated by annotation *@group(g1)*. The third reaction is the result (composite) reaction corresponding to the operator ***AND*** (*@and(g1)*) applied to the first two operands. This means that the whole group will terminate when both composed reactions are detected or timeout expires. Positive detection sends the composite event to the third reaction that is also annotated with *@group(g2)*. The fourth reaction is another (primitive) reaction that belongs to group ***g2***. The fifth reaction indicates the operator ***OR*** for the group ***g2***. The variables ***Events*** in the third and fifth reactions capture the composite events recorded as the trace of all detected messages resulting in the pattern detection.

Informally, we have something like this: *OR(e3,AND(e1,e2))*. The third reaction is the result of an ***AND*** and the fifth reaction is the result of the enclosing ***OR***. The timeout on the ***OR***-operator is smaller than that for the embedded ***AND***-operator intentionally, to test that the whole group is removed by the timeout correctly, while allowing for the same code to be used in other tests using the following inbound events.

The client code is shown below.

```
client() :-
   % Send all the test messages from a separate thread
   switch_thread(),

   % Use user-id as conversation-id (XID) for partitioning so that each user is processed sequentially
   sendMsg(user1,async,0,request,login(user1,'10.10.10.10')),
   sendMsg(user3,async,0,request,login(user3,'80.80.80.80')),
   % Wait synchronously, could have waited asynchronously instead
   java.lang.Thread.sleep(500L),
   sendMsg(user2,async,0,request,login(user2,'30.30.30.30')),
   sendMsg(user3,async,0,request,logout(user3,'80.80.80.80')),
   sendMsg(user1,async,0,request,logout(user1,'10.10.10.10')),
   sendMsg(user1,async,0,request,login(user1,'20.20.20.20')),
   java.lang.Thread.sleep(700L),
   % This is ignored due to timeout on @or that propagates to the child @and
   sendMsg(user3,async,0,request,login(user3,'90.90.90.90')),
   sendMsg(user2,async,0,request,login(user2,'40.40.40.40')),
   sendMsg(user2,async,0,request,update(user2,'30.30.30.30')),
   % This is ignored as OR will have fired
   sendMsg(user2,async,0,request,logout(user2,'30.30.30.30')).

switch_thread() :-
   sendMsgSync(XID,task,0,switch,[]),
   rcvMsg(XID,task,From,switch,[]).
```

When run, the rulebase prints the following:

```
AND detected: [[[user1,async,0,request,[logout,user1,10.10.10.10]],
[user1,async,0,request,[login,user1,20.20.20.20]]]]
Pattern detected: [[[user1,async,0,and,[[[user1,async,0,request,[logout,user1,10.10.10.10]],
[user1,async,0,request,[login,user1,20.20.20.20]]]]]]]
Pattern detected: [[[user2,async,0,request,[update,user2,30.30.30.30]]]]
```

Observe how the "update" event results in immediate pattern detection due to ***OR***. Also note that when a full ***AND*** is detected, the final composite event includes the actual structural grouping ***and*** for the ***AND*** sub-event.

To summarize, the event groups are specified using the *@group* annotations and operators like *@and* or *@or* on the reactions corresponding to the composite events. Each composite event may then again be a member of another group and so on, allowing for arbitrary nesting of event groups. Sequences of events are easily captured by reactions following one another, typically with *@timeout* annotations.

3.3. Annotations for reaction groups

This Section provides a full reference for the metadata annotations used for Event Processing extensions to Prova. Each of the annotations below apply to either ***rcvMsg*** or ***rcvMult*** primitives.

@and(GID)

This annotation is attached to an ***exit*** reaction for a reaction group. The group ID ***GID*** is the logical name of the group. At the time the reaction primitive (***rcvMsg*** or ***rcvMult***) annotated with *@and* executes, all members of the group are assumed to have already executed. This means that the creation of the group must be done by sequentially initialising the group member reactions (event channels) and then visiting the exit reaction with precise designation of the logical connective associated with the group.

The *@and* annotation requires ***ALL*** the event channels to be successfully **proved**.

For positive channels, the channel is proved if a matching event is detected (subject to multiplicity constraints if it is annotated with *@count*).

For negative channels, the channel is proved if at the time the AND reaction group is terminated (for example, by an elapsed timeout), there has been no detection of a single reaction or for *@count* annotated channels, of the required number of reactions.

For control channels, detecting events on such channel does not directly affect the successful AND group detection but only influences it by stopping, pausing, or resuming other channels in the group.

In a special case when an AND event group consists only of control channels, the group can be terminated if all control channels are terminated individually or the group timeout expires, but no pattern is ever detected.

The following example of an ***AND*** group includes two conjuncted event channels: ***login*** and ***logout***, which must arrive before the timeout of 2000 milliseconds specified on the exit channel is elapsed. The ***AND*** group sends an internal message with message type ***and*** to the engine itself, that is intercepted by the exit channel. The payload represented by the variable ***Events*** communicates the full trace of events that resulted in positive event pattern detection. Alternatively, if the timeout elapses and detection is unsuccessful with events for either of the conjuncted channels not yet detected, the group sends a different internal message with the message type ***timeout***, intercepted by the timeout channel. The timeout channels are very useful for assisting with diagnosis as well as reacting to edge conditions, as it communicates the trace of all so far received events in the payload represented by the variable ***Events***.

```
server_1(XID,User,IP) :-
   @group(g1)
   rcvMsg(XID,Protocol,From,request,login(User,IP2)) [IP2!=IP].
server_1(XID,User,IP) :-
   @group(g1)
   rcvMsg(XID,Protocol,From,request,logout(User,IP)).
server_1(XID,User,IP) :-
   @and(g1) @timeout(2000)
   rcvMsg(XID,Protocol,From,and,Events),
   println(["Pattern detected: ",Events," "]).
server_1(XID,User,IP) :-
   @and(g1)
   rcvMsg(XID,Protocol,From,timeout,Events),
   println(["Timeout occurred: ",Events," "]).
```

@count(MinCount[,MaxCount[,ignore|record|strict]])

This annotation **used in AND groups** specifies the range between ***MinCount*** and ***MaxCount*** as the required multiplicity for the matching events for this channel to be proved. The third argument talkes either of the three pre-defined values: **ignore**, **record**, or **strict**.

The **ignore** setting results in all events after ***MaxCount*** is reached to be completely ignored. Such subsequent events do not invalidate the positive detection status set after ***MinCount*** events are detected.

The **record** setting makes the events arriving after ***MaxCount*** to be recorded as part of the eventual detected pattern trace. Such subsequent events do not invalidate the positive detection status set after ***MinCount*** events are detected.

The **strict** setting results in the invalidation of the event channel and, obviously, of the containing ***AND*** pattern if one more event arrives after ***MaxCount*** events are already detected.

There are a few abbreviated forms of this annotation.

*@count(0,0,***record***)* is abbreviated as *@count(-1)*

This variant sets the multiplicity to *****, in the regular expression sense, so that an indefinite number of matching events can be received and recorded as part of the eventual detected pattern trace.

*@count(***N***,***N***,***record***)* is abbreviated as *@count(***N***,-1)*

This variant sets the multiplicity to **N**+, in the regular expression sense, so that at least **N** matching events must be received for the event channel to be proved. All received events are recorded as part of the eventual detected pattern.

*@count(***N***,***N***,***ignore***)* is abbreviated as *@count(***N***)*

This variant sets the multiplicity to **N**+, in the regular expression sense, so that at least **N** matching events must be received for the event channel to be proved. All events received after the first **N** are completely ignored.

The following example is a fairly interesting event pattern. The variable ***Basket*** is pre-initialised with a Java ***Set*** collection. For two seconds, the pattern watches for additions and removals from a basket. An indefinite number of additions is allowed (and recorded in the pattern trace) while the set ***Basket*** accumulates all added ***ID***'s. At least one removal is expected, such that the ***Basket*** does **not** contain the same ***ID***. Such removal along with all subsequent removals are recorded in the pattern trace.

```
server_1(XID,User,IP,Basket) :-
   @group(g1) @id(id1) @count(-1)
   rcvMsg(XID,Protocol,From,basket,add(User,IP,ID)) [Basket.add(ID)],
   println([User,"basket products",Basket]," ").
server_1(XID,User,IP,Basket) :-
   @group(g1) @id(id2) @count(1,-1)
   rcvMsg(XID,Protocol,From,basket,remove(User,IP,ID)) [false=Basket.contains(ID)],
   println([User,"invalid return",ID]," ").
server_1(XID,User,IP,Basket) :-
   @and(g1) @timeout(2000)
   rcvMsg(XID,Protocol,From,and,Events),
   println(["Pattern detected: ",Events," "]).
```

@group(GID)

The annotation attached to inline reactions to mark them as event channels so that they form part of a reaction group. The ***GID*** parameter is the logical name of the group. Note that there can be many active instances of groups with the same name at any moment of time, with one group created each time the engine encounters a group with a distinct logical name in any given goal or reaction to a message (which is, of course, also a goal in Prova). The type of the group is specified on the exit channel using the group type *@and* or *@or* annotations. The main ideas behind using reaction groups for event processing are discussed in Event processing using reaction groups.

@id(ID)

This annotation is the foundation of the dynamic event channels mechanism built into Prova. It assigns a unique (within the containing reaction group) logical identifier ***ID*** to the annotated event channel. As the channel acquires this identifier, it can then be ***controlled*** from other control channels, with annotations such as *@stop* or *@pause* by explicitly targeting this ***ID***. The following example hows how an ***update*** event stops (and completely removes) the first ***login*** channel.

```
server_1(XID,User,IP) :-
   @group(g1) @id(id1)
   rcvMsg(XID,Protocol,From,request,login(User,IP2)) [IP2!=IP],
   println(["Suspicious login",User,IP,IP2]," ").
server_1(XID,User,IP) :-
   @group(g1) @stop(id1)
   rcvMsg(XID,Protocol,From,request,update(User,IP)),
   println(["Update",User,IP]," ").
```

@not

Inline reactions annotated with *@not* are called negative event channels. Essentially, it means that the channel can only be detected successfully if the matching event(s) are **not** detected. Exactly how many events is determined by the presence of other annotations affecting the required multiplicity of the events in the channel. The behavior is also different depending on whether this event channel belongs to an ***AND*** or an ***OR*** group.

@not *for* AND *groups*

Consider the following example in which it may happen that other event channels apart from the one annotated with *not* are successful when timeout of the containing reaction group expires. Should it happen, the group timeout actually results in a successful event pattern detection. Conversely, the group instance fails if a matching event in the ***logout*** channel **is** detected.

```
server_1(XID,User,IP) :-
   @group(g1) @timeout(250)
   rcvMsg(XID,Protocol,From,request,login(User,IP2)) [IP2!=IP],
   println(["Suspicious login",User,IP,IP2]," ").
server_1(XID,User,IP) :-
   % This reaction succeeds immediately if all other events in the AND group arrive and the overall
group has no timeout,
   %   but will wait for the group timeout to expire before releasing result, if the group has a timeout (it
has here).
   % However, if the matching event below arrives after other reactions but before the group timeout,
the pattern is not detected.
   @group(g1) @not
   rcvMsg(XID,Protocol,From,request,logout(User,IP)),
   println(["Logout",User,IP]," ").
server_1(XID,User,IP) :-
   @and(g1) @timeout(400)
   rcvMsg(XID,Protocol,From,and,Events),
   println(["Pattern detected: ",Events," "]).
```

This behavior can be further affected by a @count annotation. If the count mode is **strict**, only the number of matching events within the range specified by the first two arguments of *@count* invalidates the group on timeout. If the mode is **ignore** or **record**, reaching the minimum number of events specified by *@count* immediately fails the ***AND*** group instance.

The example below shows how the **strict** mode sets the required multiplicity of the detected events in the ***remove*** channel to be **anything but** one.

```
server_1(XID,User,IP,Basket) :-
   @group(g1) @id(id1) @count(-1)
   rcvMsg(XID,Protocol,From,basket,add(User,IP,ID)) [Basket.add(ID)],
   println([User,"basket products",Basket]," ").
server_1(XID,User,IP,Basket) :-
   @group(g1) @id(id2) @count(1,1,strict) @not
   rcvMsg(XID,Protocol,From,basket,remove(User,IP,ID)) [false=Basket.contains(ID)],
   println([User,"invalid return",ID]," ").
```

```
server_1(XID,User,IP,Basket) :-
   @group(g1) @stop(id1,id2)
   rcvMsg(XID,Protocol,From,site,logout(User,IP)),
   println([User,"logout ",IP]," ").
server_1(XID,User,IP,Basket) :-
   @and(g1)
   rcvMsg(XID,Protocol,From,and,Events),
   println(["Pattern detected: ",Events," "]).
```

@not *for* OR *groups*

In the case of ***OR*** groups, the *@not* also requires that the matching event(s) are **not** detected in specified multiplicities but upon timeout, this channel on its own may decide whether the overall group was successful.

In the example below, if the group timeout expires and neither ***login*** not ***logout*** have been detected, the group will be successful. It will, of course, detect the pattern should the matching ***login*** be detected before timeout and the ***logout*** channel alone will be terminated should a ***logout*** be detected before timeout and a ***login***.

```
server_1(XID,User,IP) :-
   @group(g1)
   rcvMsg(XID,Protocol,From,request,login(User,IP2)) [IP2!=IP],
   println(["Suspicious login",User,IP,IP2]," ").
server_1(XID,User,IP) :-
   @group(g1) @not
   rcvMsg(XID,Protocol,From,request,logout(User,IP)),
   println(["Logout",User,IP]," ").
server_1(XID,User,IP) :-
   @or(g1) @timeout(1000)
   rcvMsg(XID,Protocol,From,or,Events),
   println(["Pattern detected: ",Events," "]).
```

Remember that ***OR*** groups can only have *@size* but not *@count* annotations. The following example shows that the *@size(N)* annotation sets a limit of N on the detected events so that if two ***login*** events arrive before the group timeout (and, of course, before a ***logout*** event, which would have resulted in immediate pattern detection), the negative channel will be terminated on its own.

```
server_1(XID,User,IP) :-
   @group(g1) @not @size(2)
   rcvMsg(XID,Protocol,From,request,login(User,IP2)) [IP2!=IP],
   println(["Suspicious login",User,IP,IP2]," ").
server_1(XID,User,IP) :-
   @group(g1)
   rcvMsg(XID,Protocol,From,request,logout(User,IP)),
   println(["Logout",User,IP]," ").
server_1(XID,User,IP) :-
   @or(g1) @timeout(1000)
   rcvMsg(XID,Protocol,From,or,Events),
   println(["Pattern detected: ",Events," "]).
server_1(XID,User,IP) :-
   @or(g1)
   rcvMsg(XID,Protocol,From,timeout,Events),
   println(["Timeout occurred: ",Events," "]).
```

@optional

A reaction annotated with *@optional* is typically used in ***AND*** groups to specify an optional contribution to the overall event pattern. This follows the idea that event patterns should not be over-specified and allow for necessary variability. Any events on this channel are entirely optional.

The annotated reaction will record the first matching event if detected during the lifetime of the containing group but will ignore all subsequent matches.

```
server_1(XID,User,IP) :-
   % Any return after the first one is ignored but the first one is optional but recorded
   @group(g1) @optional
   rcvMsg(XID,Protocol,From,request,return(User,IP)),
   println(["Return",User,IP]," ").
```

@or

This annotation is attached to an exit channel for a reaction group. The group ID ***GID*** is the logical name of the group. At the time the annotated reaction primitive (***rcvMsg*** or ***rcvMult***) executes, all members of the group are assumed to have already executed. This means that the creation of the group must be done by sequentially initialising the group member reactions (***event channels***) and then visiting the exit reaction with precise designation of the logical connective associated with the group.

The *@or* annotation requires ***EITHER*** of the event channels to be successfully **proved**.

For positive channels, the channel is proved if a matching event is detected (subject to multiplicity constraints if it is annotated with *@size*).

For negative channels, the channel is proved if at the time the OR reaction group is terminated (for example, by an elapsed timeout), there has been no detection of a single reaction or for *@size* annotated channels, of the required number of reactions.

For control channels, detecting events on such channel does not directly affect the successful OR group detection but only influences it by stopping, pausing, or resuming other channels in the group.

In a special case when an OR event group consists only of control channels, the group can be terminated if all control channels are terminated individually or the group timeout expires, but no pattern is ever detected.

The following example of an OR group detects within two seconds from the time the group instance is created either two logins from an ***IP2*** different from pre-defined ***IP*** or one logout for the same ***IP***. When the pattern is detected, the exit channel fires and the variable ***Events*** is bound to a trace of events detected in the pattern.

```
server_1(XID,User,IP) :-
   @group(g1) @size(2)
   rcvMsg(XID,Protocol,From,request,login(User,IP2)) [IP2!=IP],
   println(["Suspicious login",User,IP,IP2]," ").
server_1(XID,User,IP) :-
   @group(g1)
   rcvMsg(XID,Protocol,From,request,logout(User,IP)),
   println(["Logout",User,IP]," ").
server_1(XID,User,IP) :-
   @or(g1) @timeout(2000)
   rcvMsg(XID,Protocol,From,or,Events),
   println(["Pattern detected: ",Events," "]).
```

OR groups are the right type of groups that can be used with recurring pattern outputs using the recurring @size or recurring @timer annotations. These annotations are used for updating standard and custom aggregators and emitting the aggregations at regular event counts or time intervals.

*@pause(*ID[,ID]*)**

This annotation is part of the dynamic event channels functionality. It targets a list of controlled channels with the specified ***ID***'s and pauses them from the previous ***active state***. In a ***paused state***, the reaction is suspended, i.e., it is still part of the group (so, for example, an ***AND*** group will not succeed without that channel) but as far as the inbound events are concerned, neither of them will match the paused reaction.

This pause/resume example shows how *@pause* and *resume* work together.

@paused

This annotation is part of the dynamic event channels functionality. It makes the channel start in a ***paused state***. The channel must also be given an identifier with an *@id* annotation for another control channel to be able to resume it.

In the following example, the ***login*** channel with identifier ***id1*** starts in a paused state then if an ***update*** event is detected for the matching ***User*** and ***IP***, the ***login*** channel is resumed via a *@resume* annotation that specifically targets the channel with identifier ***id1***.

```
server_1(XID,User,IP) :-
   @group(g1) @id(id1) @paused
   rcvMsg(XID,Protocol,From,request,login(User,IP2)) [IP2!=IP],
   println(["Suspicious login",User,IP,IP2]," ").
server_1(XID,User,IP) :-
   @group(g1) @resume(id1)
   rcvMsg(XID,Protocol,From,request,update(User,IP)),
   println(["Update",User,IP]," ").
```

*@resume(ID[,ID])**

This annotation is part of the dynamic event channels functionality. It targets a list of controlled channels with the specified ***ID***'s and resumes their active state from the previous ***paused state***. In a ***paused state***, the reaction is suspended, i.e., it is still part of the group (so, for example, an ***AND*** group will not succeed without that channel) but as far as the inbound events are concerned, neither of them will match the paused reaction.

In the following example, the ***query*** events pause the ***login*** channel but the ***update*** events resume it.

```
server_1(XID,User,IP) :-
   @group(g1) @id(id1)
   rcvMsg(XID,Protocol,From,request,login(User,IP2)) [IP2!=IP],
   println(["Suspicious login",User,IP,IP2]," ").
server_1(XID,User,IP) :-
   @group(g1) @resume(id1)
   rcvMsg(XID,Protocol,From,request,update(User,IP)),
   println(["Update",User,IP]," ").
server_1(XID,User,IP) :-
   @group(g1) @pause(id1)
   rcvMsg(XID,Protocol,From,request,query(User,IP)),
   println(["Query",User,IP]," ").
server_1(XID,User,IP) :-
   @or(g1) @timeout(1000)
   rcvMsg(XID,Protocol,From,or,Events),
   println(["Pattern detected: ",Events," "]).
```

@size(Size)

This annotation **used in** *OR* **groups** sets the minimum number of events required for the annotated event channel to be proved and consequently, the ***OR*** group to be successfully detected. This example of an OR group shows the use of this annotation. In negative channels, i.e., channels annotated with *@not*, when the timeout on the ***OR*** group expires, the group could still detect a pattern, if the minimum number of events specified in the *@size* annotation is **not** reached.

@size(Size,RecurringSize,Aggregator)

This annotation is used **in OR groups** for repeated emission of pattern detections, typically for regularly outputting incremental aggregations over new events. The first output occurs when the event count reaches ***Size*** and subsequent emissions occur every ***RecurringSize*** event. The ***Aggregator*** is a Java object that can be a standard built-in or a custom aggregator capable of storing some useful information for each inbound event. In the example below, the standard ***MapCounter*** aggregator simply counts events for each key equal to *I % 5* where *I* is part of the event payload *a(I)*.

```
groupby_size() :-
   println(["==========Groupby size test=========="]),

   % Standard built-in aggregator that counts instances for each value of a key
   Counter = ws.prova.eventing.MapCounter(),
   % This reaction operates indefinitely. When the count hits 5,
   %   a copy of the current groupby map Counter is sent as part of the result,
   %   and the count is reset back to 5 (second argument of @size) and Counter is cleared.
   @group(g1) @size(5,5,Counter)
   rcvMsg(XID,async,From,inform,a(I)) [IM=I mod 5,Counter.incrementAt(IM)],
   println(["Received: ",rcvMsg(XID,async,From,inform,a(I))],"").
groupby_size() :-
   % This reaction is matched each time a new result with the groupby map for 5 input events appears.
   @or(g1)
   rcvMsg(XID,Protocol,From,or,[Results]),
   % The first element of the results is the Counter map
   Counter = Results.get(0),
   Delta = Counter.totalCount(),
   println(["Count: ",Delta]).
```

The recurring @timer annotation is similar but outputs events regularly at fixed time intervals.

*@stop(*ID[,ID]*)**

This annotation is part of the dynamic event channels functionality. It targets a list of controlled channels with specified ***ID***'s and stops them, completely removing them from memory. An event channel with such annotation is a ***control channel*** and as such does not directly participate in the event pattern detection, but indirectly affects the detection by controlling other channels. In the case of an ***AND*** group, the fact that an event channel is gone (stopped) is assumed to mean that there is no such conjunct hence the overall event pattern detection is then reduced to ensuring that all remaining event channels are successfully proved.

In the next example, the ***logout*** is a ***control channel*** that, when detected, stops the, otherwise indefinitely active, ***purchase*** channel, and consequently, results in the termination of the containing ***AND*** group and event pattern detection.

```
server_1(XID,User,IP) :-
   % Indefinitely record purchase events as part of the pattern
   @group(g1) @count(-1) @id(id1)
   rcvMsg(XID,Protocol,From,request,purchase(User,IP2)) [IP2!=IP],
   println(["Suspicious purchase",User,IP,IP2]," ").
server_1(XID,User,IP) :-
   % Any return after the first one is ignored but the first one is optional but recorded
   @group(g1) @optional
   rcvMsg(XID,Protocol,From,request,return(User,IP)),
   println(["Return",User,IP]," ").
server_1(XID,User,IP) :-
   % logout removes the id1 branch and so finishes the pattern instance as all branches are satisfied
   @group(g1) @stop(id1)
   rcvMsg(XID,Protocol,From,request,logout(User,IP)),
   println(["Logout",User,IP]," ").
server_1(XID,User,IP) :-
   @and(g1) @timeout(1000)
   rcvMsg(XID,Protocol,From,and,Events),
    println(["Pattern detected: ",Events," "]).
```

@timeout(TimeoutMilliseconds)

This annotation works for individual inline reactions, grouped event channels, and exit channels. When an inline reaction annotated with *@timeout* is created, a time limit is imposed on its lifetime, so that when the timeout expires, the inline reaction or event channel are removed completely. In the case of an event channel, the latter then does not contribute a positive detection to the containing group. In a case of an exit channel, the group instance itself has a timeout, so that that particular group instance is completely removed when the timeout expires. In this case, a special internal message is sent to the timeout channel, containing the trace of all events recorded so far in its payload.

The following example demonstrates pretty much all of the above cases. The ***logout*** event channel times out in one second on its own but the group also has a timeout of to seconds. The timeout channel receives the event traces that resulted from timed out group instances.

```
server_1(XID,User,IP) :-
    @group(g1)
    rcvMsg(XID,Protocol,From,request,login(User,IP2)) [IP2!=IP].
server_1(XID,User,IP) :-
    @group(g1) @timeout('1 sec')
    rcvMsg(XID,Protocol,From,request,logout(User,IP)).
server_1(XID,User,IP) :-
    @and(g1) @timeout('2 sec')
    rcvMsg(XID,Protocol,From,and,Events),
    println(["Pattern detected: ",Events," "]).
server_1(XID,User,IP) :-
    @and(g1)
    rcvMsg(XID,Protocol,From,timeout,Events),
    println(["Timeout occurred: ",Events," "]).
```

@timer(Delay,RecurringDelay,Processor)

This annotation is **used in OR groups** for repeated emission of pattern detections, typically for regularly outputting incremental aggregations or accumulations over new events. The first output occurs when the first timer elapses after the amount of time equal to ***Delay*** (represented as a [time period|#time_period|time period]) and subsequent emissions occur after each subsequent time interval equal to ***RecurringDelay*** (represented as a [time period|#time_period|time period]). The ***Processor*** is a Java object that is typically a standard built-in or a custom aggregator or accumulator capable of storing some useful information for each inbound event. In the example below, the standard ***MapCounter*** aggregator simply counts events for each key equal to ***I % 5***. The counter is reset every 25 milliseconds.

```
groupby_rate() :-
    println(["==========Groupby rate test=========="]),

    Counter = ws.prova.eventing.MapCounter(),
    % This reaction operates indefinitely. When the timer elapses (after 25 ms),
    %   the groupby map Counter is sent as part of the result,
    %   and the timer is reset back to the second argument of @timer.
    @group(g1) @timer(25,25,Counter)
    rcvMsg(XID,self,From,inform,a(I)) [IM=I mod 5,Counter.incrementAt(IM)],
    println(["Received: ",rcvMsg(XID,self,From,inform,a(I))],"").
groupby_rate() :-
    % This reaction is matched each time a new groupby map appears.
    @or(g1)
    rcvMsg(XID,self,From,or,[Results]),
    % The first element of the results is the Counter map
    Counter = Results.get(0),
    Delta = Counter.totalCount(),
    println(["Count: ",Delta]).
```

The second example shows how easy it is to modify the aggregation logic executed in the reaction guard to emit instead a histogram for the values in a particular field of the payload.

```
histo_rate(Min,Step) :-
    println(["==========Histogram rate test=========="]),

    Counter = ws.prova.eventing.MapCounter(),
    % This reaction operates indefinitely. When the timer elapses,
    %   the groupby map Counter is sent as part of the result,
    %   and the timer is reset back to the second argument of @timer.
    @group(g1) @timer(25,25,Counter)
    rcvMsg(XID,self,From,inform,[Payload]) [M=(Payload.get("field")-
Min)/Step,IM=M.intValue(),Counter.incrementAt(IM)],
    println(["Received: ",rcvMsg(XID,self,From,inform,[Payload])],"").
histo_rate(Min,Step) :-
    % This reaction is matched each time a new result with the histogram appears.
    @or(g1)
    rcvMsg(XID,self,From,or,[Results]),
    % The first element of the results is the Counter map
    Counter = Results.get(0),
    Delta = Counter.totalCount(),
    println(["Count: ",Delta]).
```

@vars

This annotation (only applicable in ***AND*** groups) "outjects" the local variables by name into the context of the running group instance. It also instructs the group instance to record all matching events for the annotated event channel. On top of that, when other event channels outject their variables, it matches the payloads for these channels according to (1) the matching of values of all variables with the same name and (2) the matching of any remaining variables according to constraints specified in optional *@where* annotations.

To understand why this construct is needed, consider this example of using reaction guards in a typical "followed by" event pattern.

```
server_1(XID) :-
    @group(g1)
    rcvMsg(XID,Protocol,From,request,logout(User,IP)),
    println(["Got 1"]),

    @group(g1)
    rcvMsg(XID,Protocol,From,request,login(User,IP2)) [IP2!=IP],
    println(["Got 2"]).
```

Once the first event arrives, it binds the variables ***User*** and ***IP*** to concrete values by matching them against the ***logout*** event payload. The second event channel for ***login*** events then uses a guard ***IP2!=IP*** by comparing the already bound values of ***IP*** and ***IP2***.

In the case of an inherently unordered ***AND*** group, it is not known in which order the conjuncted events will arrive so it is impossible to use guards on variables that will be potentially only known in future.

The following example based on the *@vars* annotation detects within 1.5 seconds up to four pairs of **unordered** ***login***/***logout*** pairs with matching values for ***User*** and ***IP*** variables.

```
server_1() :-
    @group(g1) @count(-1) @vars(User,IP)
    rcvMsg(q1,Protocol,From,request,login(User,IP)).
server_1() :-
    % Order of variables in @vars is not important
    @group(g1) @count(-1) @vars(IP,User)
    rcvMsg(q1,Protocol,From,request,logout(User,IP)).
server_1() :-
    @and(g1) @count(4) @timeout(1500)
    rcvMsg(q1,Protocol,From,and,Events),
    println(["Pattern detected: ",Events," "]).
server_1() :-
    @and(g1)
    rcvMsg(XID,Protocol,From,timeout,Events),
    println(["Timeout occurred: ",Events," "]).
```

This is a typical output from this pattern:

```
    % Pattern detected: [[[q1,async,0,request,[logout,user2,30.30.30.30]],
[q1,async,0,request,[login,user2,30.30.30.30]]]]
    % Pattern detected: [[[q1,async,0,request,[login,user1,10.10.10.10]],
[q1,async,0,request,[logout,user1,10.10.10.10]]]]
    % Pattern detected: [[[q1,async,0,request,[logout,user1,10.10.10.10]],
[q1,async,0,request,[login,user1,10.10.10.10]]]]
    % Timeout occurred: [[]]
```

@where(WhereExpression***)***

This annotation (working only in ***AND*** groups) complements the *@vars* annotation by executing a simple embedded expression language for the variables outjected by the event channels in the group. If the expression evaluates to ***true*** for the variables, the participating events form a tuple that is a successful event pattern detection. Note that they may be more than one tuple emitted triggered by the arrival of an event on any participating event channel.

The following gives the full ANTLR3 grammar for the *@where* expression language.

```
grammar Where;

options {
  k = 2;
  output = AST;
}

tokens {
    IN;
}

@parser::header {
  package ws.prova.parser;
}

@lexer::header {
  package ws.prova.parser;
}

expr  :  xor_expr;

par_expr:      '(' expr ')' -> expr;

xor_expr:or_expr ('xor'^ or_expr)*;

or_expr : and_expr ('or'^ and_expr)*;
```

```
and_expr:     not_expr ('and'^ not_expr)*;

not_expr:('not'^)? (par_expr | comparison | in );

comparison
    :    Identifier (Comparison^ (Identifier | T));

in
    :    Identifier 'in' '(' T (',' T)* ')' -> ^(IN Identifier T+);

fragment
Number  :    DIGIT+ ('.' DIGIT+)?;

fragment
String
    :    '\'' (~('\'' | '\n' | '\r'))* '\'';

T  :    String | Number;

Comparison
  : '!=' |  '<' | '<=' | '>=' | '>';

WhiteSpace
  : (' ' | '\t' | '\n' | '\r')+ { skip(); };

Identifier
  : CHAR (CHAR | DIGIT)*;

fragment
CHAR:    'a'..'z' | 'A'..'Z' | '_';

fragment
DIGIT:    '0'..'9';
```

A commented example of using the *@where* annotation follows.

```
server_1() :-
    @group(g1) @count(-1) @vars(User,IP1,N1)
    rcvMsg(q1,Protocol,From,request,login(User,IP1,N1)).
server_1() :-
    % Order of variables in @vars is not important
    % The @where annotation can be attached to any reaction in the group, all such constraints are
AND-ed
    @group(g1) @count(-1) @vars(User,IP2,N2) @where('IP1!=IP2 or N1<N2')
    rcvMsg(q1,Protocol,From,request,logout(User,IP2,N2)).
server_1() :-
    @and(g1) @count(6) @timeout(1500)
    rcvMsg(q1,Protocol,From,and,Events),
    println(["Pattern detected: ",Events," "]).
server_1() :-
    @and(g1)
    rcvMsg(XID,Protocol,From,timeout,Events),
    println(["Timeout occurred: ",Events," "]).
```

Time periods

If a ***time period*** is an integer number, it is assumed to be a value in milliseconds. If it is a string, it is parsed as:

TimePeriod : [Days] [Hours] [Minutes] [Seconds] [Milliseconds]

Days : (number|variable) ("d" | "day" | "days")
Hours : (number|variable) ("h" | "hour" | "hours")
Minutes : (number|variable) ("m", "min" | "minute" | "minutes")
Seconds : (number|variable) ("s", "sec" | "second" | "seconds")
Milliseconds : (number|variable) ("ms" | "msec" | "millisecond" | "milliseconds")

3.4. Lifecycle of reaction group instances

Reaction group instance creation

Each reaction group is a template from which group instances are created each time, during the evaluation of the current goal, the engine encounters an event channel with a new conversation-id ***XID***. The fact that there may exist another group instance for the same ***XID*** is irrelevant, what does matter is that during the processing of the current goal, this ***XID*** was encountered for the first time.

Reaction group instance initialisation

Once a group instance is created, the engine creates a hidden temporal reaction with current free and bound variables and then executes a Prolog ***fail*** on each event channel (***rcvMsg*** or ***rcvMult***) annotated with the same group identifier ***GID*** using *@group(*GID*)*. This ***fail*** continues the standard non-deterministic goal evaluation so that the engine visits other branches possibly containing more event channels or the exit channel for the same group. When the engine visits the exit channel, the group instance is assigned its type according to the associated annotation, *@and* or *@or*, which completes its initialisation. The initialised group instance then intercepts all events that match any of the created temporal reactions.

How many group instances can co-exist?

It is important to appreciate that there may exist quite a few active group instances at the same time. The number of active group instances is only limited by process memory and, of course, the CPU power of the machine so that it can perform matching against all existing instances for any inbound event, the task simplified by partitioning on conversation-id for any messages on the **async** protocol (that is always recommended for use in event processing in Prova), as well as by indexing of head literals.

What are the conditions for the group instance termination?

Setting aside the efficiency, a more important question is '***when do the group instances terminate?***' A typical reaction group terminates whenever the event pattern that it defines is detected, or a timeout occurs. The matter becomes, of course, more complicated if detection multiplicity or recurring pattern emission are part of the group definition. For examples, the **@vars** annotation results in continuous joins between the equivalently named variables in different channels, so that event if one tuple is detected, the group instance does not terminate until the group is terminated in some other way, for example, by the group timeout or when a control channel stops one or more channel or the whole group instance.

Group instance extension (using an infinite reaction group example)

However, there exists a way for the group instance to be "extended" by one or more new reaction, even if it already is about to be terminated, either successfully or unsuccessfully.

Consider the following complete, fairly interesting example, **rules/reloaded/stable_limit.prova**.

```
% Demonstrate detection of reaching the price level from above and staying there for a minimum
amount of time.
% Also only restart pattern processing when the price goes above the specified price level before the
timeout.
%
% This will print:
```

```
%
% Pattern detected: [[[market1,async,0,update,[price,1.4]], [market1,async,0,update,[price,1.15]]]]
% Pattern detected: [[[market2,async,0,update,[price,2.2]], [market2,async,0,update,[price,1.1]],
%                  [market2,async,0,update,[price,1.45]], [market2,async,0,update,[price,1.12]]]]

:- eval(server(1.2)).

server(Limit) :-
   % Start detection on each new market
   rcvMult(Market,Protocol,From,create,market(Market)),
   server_1(Market,Limit).

server_1(Market,Limit) :-
   @group(g1)
   rcvMsg(Market,Protocol,From,update,price(Price0)) [Price0>Limit],
   server_2(Market,Limit).
server_1(Market,Limit) :-
   @group(g1) @stop
   rcvMsg(Market,Protocol,From,destroy,market(Market)).
server_1(Market,Limit) :-
   @and(g1)
   rcvMsg(Market,Protocol,From,and,Events),
   println(["Pattern detected: ",Events]).

server_2(Market,Limit) :-
   @group(g1)
   rcvMsg(Market,Protocol,From,update,price(Price1)) [Price1<Limit],
   @group(g1) @not @timeout(1000)
   rcvMsg(Market,Protocol,From,update,price(Price2)) [Price2>Limit],
   server_2(Market,Limit).

:- eval(client()).

client() :-
   % Send all the test messages from a separate thread
   switch_thread(),

   % Use market-id as conversation-id (XID) for partitioning so that each market is processed
sequentially
   sendMsg(market2,async,0,create,market(market2)),
   sendMsg(market1,async,0,create,market(market1)),
   sendMsg(market1,async,0,update,price(1.4)),
   sendMsg(market2,async,0,update,price(2.2)),
   java.lang.Thread.sleep(200L),
   sendMsg(market2,async,0,update,price(1.1)),
   java.lang.Thread.sleep(200L),
   sendMsg(market2,async,0,update,price(1.45)),
   sendMsg(market1,async,0,update,price(1.15)),
   java.lang.Thread.sleep(1200L),

   % Second chance for market2
   sendMsg(market2,async,0,update,price(1.12)),
   java.lang.Thread.sleep(1200L).

switch_thread() :-
   sendMsgSync(XID,task,0,switch,[]),
   rcvMsg(XID,task,From,switch,[]).)
```

The code contains a test driver ***client*** that sends events to a ***server*** that, interestingly, contains a self-recursive clause ***server_2***. The test looks for the occurrences of the following sequential pattern:

1. The price is above the ***Limit***.
2. The price is below the ***Limit***.
3. The price stays below the ***Limit*** for at least a specified interval of time.

Now, a lot of things can go wrong here.

1. We have to make sure that a new detection does not start each time the updated price is **still** above ***Limit***.
2. We have to make sure that we restart detection if the price goes back up too soon, before the specified interval of time expires.

The first ***server_1*** clause guarantees the first requirement. As ***rcvMsg*** is used here with multiplicity one, only the very first price update with ***price>Limit*** will start the detection. Does not this then contradict the second requirement? No, because of the second reaction in the ***server_2*** clause.

This reaction is annotated with both *@not* and *@timeout*. If the matching event (price above ***Limit***) arrives before the timeout, the ***AND*** group should have been normally terminated. However, due to recursion to the first reaction in ***server_2*** again, the reaction group life is extended so that the wait begins for the price to dip down again under ***Limit***.

When a group instance gets extended?

IF

1. a reaction group instance receives a matching event in one of its group channels
2. AND the conditions are met for the group instance to either successfully detect an event pattern or be unsuccessfully terminated
3. AND the evaluation of the continuation subgoal encounters reaction(s) for the same logical group and conversation-id

THEN

1. the group completion or termination is canceled,
2. one or more new temporal reactions are created and initialised as part of this group instance,
3. the lifetime of the reaction group instance is thereby extended.

In the above example, the group instance for logical group ***g1*** and conversation-id ***Market*** is extended in the self-recursive clause ***server_2*** and will continue looking for the time interval when the price stays low continuously.

3.5. Context variables with equality and WHERE constraints in AND groups

Reaction groups (see Structure of reaction groups) is a mechanism for defining multiple reactions in a common context. Associating common semantics, timeout, aggregations are typical contextual enrichments of primitive message processing. Using variables and constraints is another context-wide functionality of reaction groups in Prova.

Each reaction in an *@and* event processing group can have a *@vars* annotation that lists all Prova variables occurring in the reaction pattern or its context to be made available to constraint solving based on more than one reaction in the group. This is particularly useful when we are looking for *@and* patterns that include undefined orderings of events with specific constraints between them. Note that in the case of a "followed by" operator, earlier reactions set the appropriate variables and a simple guard on subsequent reaction is sufficient for specifying constraints.

The annotation *@vars* works in such a way that equational constraints do not need to be explicit, i.e., as far as the syntax is concerned, it is sufficient to reuse the same names of one or more variables in multiple reactions for these variables to be equated in any detected sequence of events. The test ***and_vars.prova*** demonstrates this functionality. It detects in the inbound stream one unordered login/logout pair with matching ***User*** and ***IP***.

```
% Demonstrate an @and group with equational constraints using context variables.
% Only the first pair is detected since reactions for login and logout have default multiplicity of one.
```

```
:- eval(server()).

server() :-
   server_1().

server_1() :-
   @group(g1) @vars(User,IP)
   rcvMsg(q1,Protocol,From,request,login(User,IP)).
server_1() :-
   @group(g1) @vars(User,IP)
   rcvMsg(q1,Protocol,From,request,logout(User,IP)).
server_1() :-
   @and(g1) @timeout(2000)
   rcvMsg(q1,Protocol,From,and,Events),
   println(["Pattern detected: ",Events," "]).

:- eval(client()).

client() :-
   % Send all the test messages from a separate thread
   switch_thread(),

   % Send all events on the same conversation-id for sequential processing
   sendMsg(q1,async,0,request,login(user1,'10.10.10.10')),
   % Wait synchronously, could have waited asynchronously instead
   java.lang.Thread.sleep(500L),
   sendMsg(q1,async,0,request,login(user2,'30.30.30.30')),
   java.lang.Thread.sleep(700L),
   sendMsg(q1,async,0,request,logout(user1,'10.10.10.10')),
   % If this message arrives before the previous one, it removes the logout reaction above, preventing the
chance
   %    of a logout matching (user1,10.10.10.10) being detected.
   % The @count annotation must be used to allow for reaction to stay and accumulate the matches (see
and_multi_repeated.prova).
   sendMsg(q1,async,0,request,logout(user2,'30.30.30.30')),
   sendMsg(q1,async,0,request,login(user1,'20.20.20.20')),
   java.lang.Thread.sleep(1500L),
   sendMsg(q1,async,0,request,login(user2,'40.40.40.40')).

switch_thread() :-
   sendMsgSync(XID,task,0,switch,[]),
   rcvMsg(XID,task,From,switch,[]).
```

This example will print:

```
Pattern detected: [[[q1,async,0,request,[login,user1,10.10.10.10]],
[q1,async,0,request,[logout,user1,10.10.10.10]]]]
```

The test ***and_multi_vars.prova*** adds *@count* annotations to allow for more than one detection to be emitted from one active pattern group instance.

```
% Demonstrate an @and group with equational constraints on group context variables.
% During 1500 ms, detect up to 4 unordered pairs of login and logout with matching User and IP.
% Note that neither User or IP are initially known.

:- eval(server()).

server() :-
   % This construct is useful for @and groups where a variable should be shared across multiple
reactions.
```

```
    server_1().

server_1() :-
    @group(g1) @count(-1) @vars(User,IP)
    rcvMsg(q1,Protocol,From,request,login(User,IP)).
server_1() :-
    % Order of variables in @vars is not important
    @group(g1) @count(-1) @vars(IP,User)
    rcvMsg(q1,Protocol,From,request,logout(User,IP)).
server_1() :-
    @and(g1) @count(4) @timeout(1500)
    rcvMsg(q1,Protocol,From,and,Events),
    println(["Pattern detected: ",Events," "]).
server_1() :-
    @and(g1)
    rcvMsg(XID,Protocol,From,timeout,Events),
    println(["Timeout occurred: ",Events," "]).

:- eval(client()).

client() :-
    % Send all the test messages from a separate thread
    switch_thread(),

    % Process all messages on the same partition
    sendMsg(q1,async,0,request,login(user1,'10.10.10.10')),
    % Wait synchronously, could have waited asynchronously instead
    java.lang.Thread.sleep(500L),
    sendMsg(q1,async,0,request,logout(user2,'30.30.30.30')),
    java.lang.Thread.sleep(700L),
    sendMsg(q1,async,0,request,login(user2,'30.30.30.30')),
    sendMsg(q1,async,0,request,logout(user1,'10.10.10.10')),
    sendMsg(q1,async,0,request,login(user1,'20.20.20.20')),
    sendMsg(q1,async,0,request,login(user1,'10.10.10.10')),
    java.lang.Thread.sleep(1500L),
    sendMsg(q1,async,0,request,login(user1,'10.10.10.10')),
    sendMsg(q1,async,0,request,login(user2,'40.40.40.40')).

switch_thread() :-
    sendMsgSync(XID,task,0,switch,[]),
    rcvMsg(XID,task,From,switch,[]).
```

This will print:

```
    Pattern detected: [[[q1,async,0,request,[logout,user2,30.30.30.30]],
[q1,async,0,request,[login,user2,30.30.30.30]]]]
    Pattern detected: [[[q1,async,0,request,[login,user1,10.10.10.10]],
[q1,async,0,request,[logout,user1,10.10.10.10]]]]
    Pattern detected: [[[q1,async,0,request,[logout,user1,10.10.10.10]],
[q1,async,0,request,[login,user1,10.10.10.10]]]]
    Timeout occurred: [[]]
```

Finally, the annotation *@where* can carry expressions in a simple constraint language that are used for verifying constraints involving different context variables from multiple reactions. You can use various comparison operators as well as use logical connectives for building more complex logical expressions. The test where.prova shows how it all works together.

```
% Demonstrate an @and group with WHERE constraints on group context variables.
% During 1500 ms, detect up to 6 unordered pairs of login and logout with matching User and IP.
% Note that neither User or IP are initially known.
```

```
:- eval(server()).

server() :-
    % This construct is useful for @and groups where a variable should be shared across multiple
reactions.
    server_1().

server_1() :-
    @group(g1) @count(-1) @vars(User,IP1,N1)
    rcvMsg(q1,Protocol,From,request,login(User,IP1,N1)).
server_1() :-
    % Order of variables in @vars is not important
    % The @where annotation can be attached to any reaction in the group, all such constraints are
AND-ed
    @group(g1) @count(-1) @vars(User,IP2,N2) @where('IP1!=IP2 or N1<N2')
    rcvMsg(q1,Protocol,From,request,logout(User,IP2,N2)).
server_1() :-
    @and(g1) @count(6) @timeout(1500)
    rcvMsg(q1,Protocol,From,and,Events),
    println(["Pattern detected: ",Events," "]).
server_1() :-
    @and(g1)
    rcvMsg(XID,Protocol,From,timeout,Events),
    println(["Timeout occurred: ",Events," "]).

:- eval(client()).

client() :-
    % Send all the test messages from a separate thread
    switch_thread(),

    % Process all messages on the same partition
    sendMsg(q1,async,0,request,login(user1,'10.10.10.10',12)),
    % Wait synchronously, could have waited asynchronously instead
    java.lang.Thread.sleep(500L),
    sendMsg(q1,async,0,request,logout(user2,'30.30.30.30',5)),
    java.lang.Thread.sleep(700L),
    sendMsg(q1,async,0,request,login(user2,'30.30.30.30',3)),
    sendMsg(q1,async,0,request,logout(user1,'10.10.10.10',15)),
    sendMsg(q1,async,0,request,login(user1,'20.20.20.20',17)),
    sendMsg(q1,async,0,request,login(user1,'30.30.30.30',18)),
    java.lang.Thread.sleep(1500L),
    sendMsg(q1,async,0,request,login(user1,'10.10.10.10',20)),
    sendMsg(q1,async,0,request,login(user2,'40.40.40.40',7)).

switch_thread() :-
    sendMsgSync(XID,task,0,switch,[]),
    rcvMsg(XID,task,From,switch,[]).
```

This will print:

```
    Pattern detected: [[[q1,async,0,request,[logout,user2,30.30.30.30,5]],
[q1,async,0,request,[login,user2,30.30.30.30,3]]]]
    Pattern detected: [[[q1,async,0,request,[login,user1,10.10.10.10,12]],
[q1,async,0,request,[logout,user1,10.10.10.10,15]]]]
    Pattern detected: [[[q1,async,0,request,[logout,user1,10.10.10.10,15]],
[q1,async,0,request,[login,user1,20.20.20.20,17]]]]
    Pattern detected: [[[q1,async,0,request,[logout,user1,10.10.10.10,15]],
[q1,async,0,request,[login,user1,30.30.30.30,18]]]]
```

```
Timeout occurred: [[]]
```

3.6. Dynamic event channels

Dynamic event channels is the mechanism for selectively controlling event channels through the arrival of events on other control channels.

This functionality relies on an annotation *@id(*ID*)* that associates a (unique within the group) name with an event channel. Once we have a way to refer to a specific channel, we can use events arriving on ***control channels***to affect the named channel(s). There are quite a few natural things we can do. We can permanently stop listed channels by using a*@stop(*IDLIST*)* annotation on a control channel. We also can pause and resume detection of events on channels with **@pause(IDLIST)** and *@resume(*IDLIST*)* on control channels. A paused event channel ignores any events that would have matched the paused reaction. Finally, a channel can start in a paused state (denoted by the *@paused* annotation), in which case, it will take an event arrival on a control *@resume(*IDLIST*)*channel to unpause the initially paused channel. Here is an example **rules/reloaded/or_paused.prova** demonstrating the described functionality. Obviously, we cannot show all combination here but the ProvaMetadataTest.java included in the Prova code runs an extensive suite of tests demonstrating a large number of situations.

```
% Demonstrate channels that are initially paused with @paused and then resumed with @resume.
%
:- eval(server()).

server() :-
   % Start detection on each new login
   rcvMult(XID,Protocol,From,request,login(User,IP)),
   server_1(XID,User,IP).

server_1(XID,User,IP) :-
   @group(g1) @id(id1) @paused
   rcvMsg(XID,Protocol,From,request,login(User,IP2)) [IP2!=IP],
   println(["Suspicious login",User,IP,IP2]," ").
server_1(XID,User,IP) :-
   @group(g1)
   rcvMsg(XID,Protocol,From,request,logout(User,IP)),
   println(["Logout",User,IP]," ").
server_1(XID,User,IP) :-
   @group(g1) @resume(id1)
   rcvMsg(XID,Protocol,From,request,update(User,IP)),
   println(["Update",User,IP]," ").
server_1(XID,User,IP) :-
   @or(g1) @timeout(1000)
   rcvMsg(XID,Protocol,From,or,Events),
   println(["Pattern detected: ",Events," "]).

:- eval(client()).

client() :-
   % Send all the test messages from a separate thread
   switch_thread(),

   % Use user-id as conversation-id (XID) for partitioning so that each user is processed sequentially
   sendMsg(user1,async,0,request,login(user1,'10.10.10.10')),
   java.lang.Thread.sleep(200L),
   sendMsg(user2,async,0,request,login(user2,'30.30.30.30')),
   java.lang.Thread.sleep(300L),
   sendMsg(user1,async,0,request,logout(user1,'10.10.10.10')),
   sendMsg(user1,async,0,request,login(user1,'20.20.20.20')),
   sendMsg(user2,async,0,request,login(user2,'40.40.40.40')),
   sendMsg(user2,async,0,request,update(user2,'30.30.30.30')),
   sendMsg(user2,async,0,request,login(user2,'50.50.50.50')).

switch_thread() :-
   sendMsgSync(XID,task,0,switch,[]),
   rcvMsg(XID,task,From,switch,[]).
```

The above example outputs:

```
Logout user1 10.10.10.10
Suspicious login user2 30.30.30.30 40.40.40.40
Update user2 30.30.30.30
Pattern detected: [[[user1,async,0,request,[logout,user1,10.10.10.10]]]]
Suspicious login user2 30.30.30.30 50.50.50.50
Suspicious login user2 40.40.40.40 50.50.50.50
Pattern detected: [[[user2,async,0,request,[update,user2,30.30.30.30]],
[user2,async,0,request,[login,user2,50.50.50.50]]]]
```

Note that the pair of logins from 30.30.30.30 followed by 40.40.40.40 is not the pattern as detection of further logins is initially paused until an 'update' event is detected.

3.7. EPTS Language Working Group

This Section includes additional documents produced for Event Processing Technical Society Language Working Group.

3.7.1. Flower delivery example

This is a work on progress on delivering a Prova EP implementation for the use case proposed by Opher Etzion and Peter Niblett for the upcoming book "Event Processing in Action".

The working project containing many more other tests can be checked out from https://mandarax.svn.sourceforge.net/svnroot/mandarax/mule-prova-agents2/trunk. The project relies on maven2 and the general instructions to get you up to speed are available here: http://www.prova.ws/downloads.php. You do not actually need the Prova project itself, just the project mentioned before. To run the test, find and run **Prova3FlowerDeliveryTest.java**.

The first part of the use case as a set of distributed agents using the Prova Mule ESB is available. This is a test Prova3FlowerDeliveryTest.java that it is possible to run by checking out the mule-prova-agents project in Prova Subversion. The implementation highlights that

- it is easy to distribute Prova agents using a variety of transports provided by the Mule ESB;
- the example will use JMS, both topics, for stores and drivers broadcasts, and queues, for direct communication and responding to senders;
- it is easy to run the whole application as a test with verifiable requirements;
- the Prova agents are not aware of the transports used and the latter could be changed (for example, from JMS to TCP if required);
- Prova is a practical rather than a purist language that allows one to fully use any Java libraries to complement and extend its functionality, for example, **ws.prova.eventing.SortedAccumulator** is used for accumulating and sorting the arriving records in real time (see the manual assignment mode for a flower store below);
- reaction groups can be used for modelling protocols with multiple choice and timeouts, as well as event pattern detection.

The conversation runs on conversation-id ***RequestId***, which "pins" all the conversation to a single thread, so that no synchronization is required. See details in the **async** protocol in Concurrent reactive messaging.

So far there are two drivers and two stores. The agents are connected to the MuleESB with the following critical part of the configuration.

mule-prova3-config-flower-delivery.xml

```
	<jms:endpoint name="store1" queue="store1" />
	<jms:endpoint name="store2" queue="store2" />
	<jms:endpoint name="stores" topic="stores" />
	<jms:endpoint name="driver1" queue="driver1" />
	<jms:endpoint name="driver2" queue="driver2" />
	<jms:endpoint name="drivers" topic="drivers" />

	<model name="mule-prova-flower-delivery">
	    <!-- All the actual Java components for these services are defined in the associated Spring
configuration -->
	    <service name="Store1Agent">
	        <inbound>
	            <jms:inbound-endpoint topic="stores" />
	            <jms:inbound-endpoint queue="store1" />
	        </inbound>
	        <component>
	            <spring-object bean="store1Service" />
	        </component>
	    </service>
	    <service name="Store2Agent">
	        <inbound>
	            <jms:inbound-endpoint topic="stores" />
	            <jms:inbound-endpoint queue="store2" />
	        </inbound>
	        <component>
	            <spring-object bean="store2Service" />
	        </component>
	    </service>
	    <service name="Driver1Agent">
	        <inbound>
	            <jms:inbound-endpoint topic="drivers" />
	            <jms:inbound-endpoint queue="driver1" />
	        </inbound>
	        <component>
	            <spring-object bean="driver1Service" />
	        </component>
	    </service>
	    <service name="Driver2Agent">
	        <inbound>
	            <jms:inbound-endpoint topic="drivers" />
	            <jms:inbound-endpoint queue="driver2" />
	        </inbound>
	        <component>
	            <spring-object bean="driver2Service" />
	        </component>
	    </service>
	    <service name="MasterAgent">
	        <component>
	            <spring-object bean="masterService" />
	        </component>
	    </service>
	</model>
```

The test runs until the drivers get two assignments each, which is expected given the static data and sent orders. The test is heavily concurrent so that each agent runs in parallel, and processing for different orders inside each store is also fully concurrent at low level.

This is a rulebase included from each driver agent.

commons_driver.prova

```
rcvMsg(XID,Esb,From,init,[]) :-
   gps_coordinates(VanId,Longitude,Latitude),
   println(["[",VanId,"] starting at (",Longitude,",",Latitude,")"]),

   % Third argument (destination) is the "stores" topic
   sendMsg(VanId,esb,stores,update,gps_coordinates(Latitude,Longitude)),

   van(VanId).

% Send coordinates updates regularly (note the tail recursion)
van(VanId) :-
   java.lang.Thread.sleep(5000L),
   gps_coordinates(VanId,Latitude,Longitude),
   % Third argument (destination) is the "stores" topic
   sendMsg(VanId,esb,stores,update,gps_coordinates(Latitude,Longitude)),
   van(VanId).

rcvMsg(RequestId,Protocol,StoreId,request,request_bid(RequestId,StoreId,StoreRegion)) :-
   gps_coordinates(VanId,Latitude,Longitude),
   sendMsg(RequestId,esb,StoreId,response,delivery_bid(VanId,RequestId)).

rcvMsg(RequestId,Protocol,StoreId,request,assignment(RequestId,StoreId)) :-
   sendMsg(XID,esb,"vm://global",job_completed,[]).

% A testing harness catch-all reaction for printing all incoming messages.
rcvMsg(XID,Protocol,From,Performative,[X|Xs]) :-
   gps_coordinates(VanId,_,_),
   println(["[",VanId,"]: ",Performative," for ",XID,": ",[From,X|Xs]]).
```

This is the code for driver1.

driver1.prova

```
:- eval(consult("flower_delivery/commons_driver.prova")).

% Fake constant position for now
gps_coordinates(driver1,30.0,30.0).
```

This is the code for driver2.

driver2.prova

```
:- eval(consult("flower_delivery/commons_driver.prova")).

% Fake constant position for now
gps_coordinates(driver2,50.0,50.0).
```

This is the common included rulebase for each store.

commons_store.prova

```
:- eval(consult("utils2.prova")).

rcvMsg(XID,Esb,From,init,[]) :-
   store_minrank(StoreId,MinRank),
   println(["[",StoreId,"] starting with minimum rank of ",MinRank]),
    checkin_drivers().

send_orders() :-
    % Send all the test messages from a separate thread
    switch_thread(),

    sendMsg(req1,async,0,request,delivery_request(req1)),
    % Wait synchronously, could have waited asynchronously instead
    java.lang.Thread.sleep(500L),
    sendMsg(req2,async,0,request,delivery_request(req2)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Phase 0, missed in the Spec %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Collect at least one update for each driver prior to proceeding with requests
checkin_drivers() :-
    @group(init_drivers) @count(2)
    rcvMsg(VanId,Protocol,VanId,update,gps_coordinates(Latitude,Longitude)).
checkin_drivers() :-
    @and(init_drivers) @timeout('10 sec')
    rcvMsg(VanId,Protocol,VanId,and,[Updates]),
    % All drivers have sent updates within the specified timeout
    % Go-Go-Go! Ready to send orders now
    send_orders().
checkin_drivers() :-
    @and(init_drivers)
    rcvMsg(VanId,Protocol,VanId,timeout,[Updates]),
    % Not all drivers have sent updates within the specified timeout, for now just print this
    println(["Not all drivers have sent updates within the specified timeout"]).

%%%%%%%%%%%
% Phase 1 %
%%%%%%%%%%%

% Enrich delivery requests, use partitioning on RequestId for concurrency
rcvMsg(RequestId,Protocol,From,request,delivery_request(RequestId)) :-
    store_minrank(StoreId,MinRank),
    sendMsg(RequestId,Protocol,From,request,enriched_delivery_request(RequestId,StoreId,MinRank)).
% Update van location
rcvMsg(VanId,Protocol,VanId,update,gps_coordinates(Latitude,Longitude)) :-
    Region = ws.prova.flower.Helper.translate(Latitude,Longitude),
    van_region(VanId,Region0),
    Region!=Region0,
    retract(van_region(VanId,Region0)),
    assert(van_region(VanId,Region)).
% Select drivers matching the order and send bid requests
rcvMsg(RequestId,Protocol,From,request,enriched_delivery_request(RequestId,StoreId,MinRank)) :-
    % Note that this search may terminally fail
    van_rank(VanId,Rank),
    MinRank<=Rank,
    van_region(VanId,Region),
    store_region(StoreId,StoreRegion),
    near(Region,StoreRegion),
    % A driver that satisfies criteria, internal protocol used for testing
    sendMsg(RequestId,esb,VanId,request,request_bid(RequestId,StoreId,StoreRegion)).

%%%%%%%%%%%
% Phase 2 %
```

```
%%%%%%%%%%%

rcvMsg(RequestId,Protocol,From,request,enriched_delivery_request(RequestId,StoreId,MinRank)) :-
   store_mode(StoreId,automatic),
   process_automatic(RequestId,StoreId).
rcvMsg(RequestId,Protocol,From,request,enriched_delivery_request(RequestId,StoreId,MinRank)) :-
   store_mode(StoreId,manual),
   process_manual(RequestId,StoreId).

process_automatic(RequestId,StoreId) :-
   @group(bids)
   rcvMsg(RequestId,Protocol,VanId,response,delivery_bid(VanId,RequestId)),
   sendMsg(RequestId,esb,VanId,request,assignment(RequestId,StoreId)).
process_automatic(RequestId,StoreId) :-
   % Exit channel to define the operator, timeout 2 seconds for fast testing, limit to one result
   @and(bids) @timeout(2000) @count(1)
   rcvMsg(RequestId,Protocol,From,and,[Events]).
process_automatic(RequestId,StoreId) :-
   % Timeout channel for no bids in the message payload, full consumption, no other branches
   @and(bids)
   rcvMsg(RequestId,Protocol,Sender,timeout,[]),
   alert_destination(no_bids,Destination),
   sendMsg(RequestId,esb,Destination,alert,no_bids(RequestId)).

process_manual(RequestId,StoreId) :-
   Acc = ws.prova.eventing.SortedAccumulator(),
   @group(bids) @timer(2000,2000,Acc)
   rcvMsg(RequestId,Protocol,VanId,response,delivery_bid(VanId,RequestId)),
   van_rank(VanId,Rank),
   Acc.processAt(Rank,VanId).
process_manual(RequestId,StoreId) :-
   % Exit channel to define the operator
   % Accept results after just a single (otherwise, recurring every 2000 ms) output
   @or(bids) @count(1)
   rcvMsg(RequestId,Protocol,From,or,[Results]),
   Acc = Results.get(0),
   Top = Acc.highest(5),
   sendMsg(RequestId,esb,StoreId,request,top(RequestId,StoreId,Top)).

rcvMsg(RequestId,Protocol,From,request,top(RequestId,StoreId,Top)) :-
   % Just choose the lowest entry from the top and send assignment
   VanId = Top.get(0),
   sendMsg(RequestId,esb,VanId,request,assignment(RequestId,StoreId)).

% A testing harness catch-all reaction for printing all incoming messages.
rcvMsg(XID,Protocol,From,Performative,[X|Xs]) :-
   store_minrank(StoreId,_),
   println(["[",StoreId,"]: ",Performative," for ",XID,": ",[From,X|Xs]]).
```

This is the code for store1.

store1.prova

```
:- eval(consult("flower_delivery/commons_store.prova")).

%%%%%%%%%%%%%%%%%%%
% Static test data %
%%%%%%%%%%%%%%%%%%%
```

```
store_minrank(store1,4).

store_region(store1,region2).

van_region(driver1,noregion).
van_region(driver2,noregion).

van_rank(driver1,5).
van_rank(driver2,2).

alert_destination(no_bids,store1).
alert_destination(no_bids,store_manager1).

near(region1,region2).
near(region1,region4).
near(region2,region1).
near(region2,region4).
near(region4,region1).
near(region4,region2).
```

This is the code for store1.

store1.prova

```
:- eval(consult("flower_delivery/commons_store.prova")).

%%%%%%%%%%%%%%%%%%%%
% Static test data %
%%%%%%%%%%%%%%%%%%%%

store_minrank(store2,1).

store_region(store2,region4).

van_region(driver1,noregion).
van_region(driver2,noregion).

van_rank(driver1,5).
van_rank(driver2,2).

alert_destination(no_bids,store2).
alert_destination(no_bids,store_manager2).

near(region1,region2).
near(region1,region4).
near(region2,region1).
near(region2,region4).
near(region4,region1).
near(region4,region2).
```

master.prova

```
% A master agent that initiates the primary conversation

:-eval(master()).

master() :-
   sendMsg(XID,esb,stores,init,[]),
   sendMsg(XID,esb,drivers,init,[]),
   println(["Master started"]).
```

The actual Java test **Prova3FlowerDeliveryTest.java** initializes Mule, which starts a local ActiveMQ JMS and all the agents and awaits for four confirmations from the drivers.

Prova3FlowerDeliveryTest.java

```
package test.org.mule.prova.agents;

import javax.jms.JMSException;
import javax.jms.ObjectMessage;

import org.junit.Test;
import org.mule.api.MuleException;
import org.mule.api.MuleMessage;
import org.mule.module.client.MuleClient;

import ws.prova.kernel2.ProvaList;

public class Prova3FlowerDeliveryTest extends AbstractBasicTest {

    public void setUp() {
      setUp("mule-prova3-config-flower-delivery.xml");
    }

    @Test
    public void testConnect() throws MuleException, JMSException {
        MuleClient client = new MuleClient();

        // There will be four signals from driver1 sent when it receives two assignments each from two
stores
        for( int i=0; i<4; i++ ) {
            MuleMessage inbound = client.request("vm://global", 1000000);
            assertNotNull(inbound);
            ProvaList rMsg = null;
            if( inbound.getPayload() instanceof ObjectMessage ) {
                rMsg = (ProvaList) ((ObjectMessage) inbound.getPayload()).getObject();
            } else {
                rMsg = (ProvaList) inbound.getPayload();
            }
            assertEquals(rMsg.performative(),"job_completed");
        }
    }

}
```

This is the output trace from running the test. Note that the precise ordering may be different due to concurrency. Also observe that all four assignments go to ***driver1*** that has the highest rank.

```
Master started
[store1] starting with minimum rank of 4
[driver2] starting at (50.0,50.0)
[store2] starting with minimum rank of 1
[driver1] starting at (30.0,30.0)
[store2]: update for driver1: [driver1,gps_coordinates,30.0,30.0]
[store1]: update for driver1: [driver1,gps_coordinates,30.0,30.0]
[store2]: update for driver2: [driver2,gps_coordinates,50.0,50.0]
[store1]: update for driver2: [driver2,gps_coordinates,50.0,50.0]
[store2]: request for req1: [0,delivery_request,req1]
[store1]: request for req1: [0,delivery_request,req1]
[store1]: request for req1: [0,enriched_delivery_request,req1,store1,4]
[store2]: request for req1: [0,enriched_delivery_request,req1,store2,1]
```

```
[driver1]: request for req1: [store2,request_bid,req1,store2,region4]
[driver1]: request for req1: [store1,request_bid,req1,store1,region2]
[driver2]: request for req1: [store2,request_bid,req1,store2,region4]
[store2]: bid received: [delivery_bid,driver1,req1]
[store2]: response for req1: [driver1,delivery_bid,driver1,req1]
[store2]: bid received: [delivery_bid,driver2,req1]
[store2]: response for req1: [driver2,delivery_bid,driver2,req1]
[store1]: response for req1: [driver1,delivery_bid,driver1,req1]
[driver1]: request for req1: [store1,assignment,req1,store1]
[store2]: request for req2: [0,delivery_request,req2]
[store1]: request for req2: [0,delivery_request,req2]
[store1]: request for req2: [0,enriched_delivery_request,req2,store1,4]
[store2]: request for req2: [0,enriched_delivery_request,req2,store2,1]
[driver1]: request for req2: [store2,request_bid,req2,store2,region4]
[driver1]: request for req2: [store1,request_bid,req2,store1,region2]
[driver2]: request for req2: [store2,request_bid,req2,store2,region4]
[store2]: bid received: [delivery_bid,driver1,req2]
[store2]: response for req2: [driver1,delivery_bid,driver1,req2]
[store2]: bid received: [delivery_bid,driver2,req2]
[store2]: response for req2: [driver2,delivery_bid,driver2,req2]
[store1]: response for req2: [driver1,delivery_bid,driver1,req2]
[driver1]: request for req2: [store1,assignment,req2,store1]
[store2]: request for req1: [store2,top,req1,store2,[driver1, driver2]]
[driver1]: request for req1: [store2,assignment,req1,store2]
[store2]: request for req2: [store2,top,req2,store2,[driver1, driver2]]
[driver1]: request for req2: [store2,assignment,req2,store2]
```

3.8. Glossary

This is a glossary of terms used in the Event Processing extensions to the Prova rule language.

control channel

An event channel that is annotated with *@pause*, *@resume* or *@stop* annotations.

event channel

An inline reaction that is a member of a reaction group.

exit channel

An event channel that intercepts the internal message sent by a reaction group when an event pattern is successfully detected. The required annotation on this channel includes its logical grouping, either *@and* or *@or* as well as optional timeout or multiplicity conditions with *@timeout* and *@count*, respectively. See the example of an ***AND*** group here.

global reaction

A rule with ***rcvMsg*** as the head literal. It has the same life scope as the engine instance running the rulebase. When the engine detects an inbound message that matches the pattern in the arguments of this ***rcvMsg***, the rule body is executed just as it happens when a goal matching the head literal is executed. See the details and example in global reaction rules.

inline reaction

A literal for a message receiving primitive ***rcvMsg*** (or ***rcvMult***). When it is executed by the engine, the latter creates a temporal hidden rule containing a closure of all remaining literals in the current goal with all context variables baked into it and immediately fails (in the Prolog sense), allowing for other non-deterministic branches to be visited. When subsequently the engine detects an inbound message matching the pattern of the arguments inside ***rcvMsg***, the message is accepted and the aforementioned closure is executed. There are many variations related to the way the inbound messages are accepted by inline reactions, see full details in inline reaction rules. Inline reactions can form groups which is the Prova way of doing Event Processing, see the discussion on reaction groups. Note that the actual reaction can be executed on a different thread from the one that executes the message receiving literal, see the Section on concurrent reactive messaging.

positive channel

An event channel that is not annotated with *@not* and is not a control channel.

negative channel

An event channel that is annotated with *@not* and is not a control channel.

reaction group

This construct forms the basis of the event processing functionality in Prova. The idea is to provide a common context for more than one concurrently enabled inline reaction (see the extensive discussion here).

Membership of event channels in a reaction group is indicated using the annotation @group and assigned a group id ***GID***, unique within the containing rulebase. The exit channels of type @and and @or define an internal callback for the results of pattern detection published each time the reaction group is **proved**. The group is considered as **proved** when the conditions depending on its type and a variety of annotations providing further semantic fine-tuning are verified. For example, an ***AND*** group may require two inline reactions to arrive before a specified timeout for the event pattern to be detected. There are settings for which reaction groups emit multiple event pattern detections, notably when using the @vars annotation in ***AND*** groups and for emitting recurring aggregates using @size or @timer in ***OR*** groups.

It is important to understand that a reaction group, as part of the containing rulebase run by the Prova engine, is only a ***template*** for reaction group ***instances***. Each of these instances is created and instantiated anew each time the engine visits an inline reaction ***rcvMsg*** (or ***rcvMult***) with a unique ***GID*** for the current goal. This creates a group instance that becomes fully instantiated once the exit channel reaction is visited as well.

timeout channel

An event channel that intercepts internal messages sent by the containing reaction group when the timeout on the group expires. See this example of an AND group.