

Law Tester: The User and Implementation Manual

Wenxuan Zhang, Constantin Serban, Naftaly Minsky

10/16/05

1 Introduction

The law tester is a light off-line tool intended to help the debugging of laws during the development process. During runtime execution, the deployment and execution of a law takes place in every controller on behalf of every adopted agent. This deployment scheme renders the process of debugging and testing difficult. The law tester addresses exactly this issue: it allows the execution and debugging of a law as part of a stand-alone program, with dynamic context. The law tester covers two aspects:

- The syntactic verification of the laws according to the grammar of either Prolog or Java, and
- The runtime evaluation of each type of event in a given context.

The law tester is designed to provide preliminary testing of laws; the tester itself does not prove that a law is (semantically) valid - usually a comprehensive testing in a large-scale LGI community is also necessary.

2 Law testing tutorial

The law tester is designed as an interactive tool that receives commands from the standard input. There are two steps involved in running the tester: 1) the loading of the law, and 2) the dynamic evaluation of events. This section presents these steps.

2.1 Dynamic law loading

2.1.1 Starting the tester

In order to start the law tester, obtain a console and issue the following command:

```
⌘ java moses.Tester
```

Please input the law file name with path (or URL):

After starting, the law tester displays the above message and waits for the user to input the destination law file. This can be either an URL or a local file:

Please input the law file name with path (or URL):

somelawfile.java1

or

Please input the law file name with path (or URL):

http://server/somelawfile.law

2.1.2 Preamble validation

Any LGI law has two parts, the preamble and the body. The preamble which is language independent is checked first upon loading. Whenever a syntax error occurs, its location is reported to the user. An example of such error, when encountered in an alias declaration is reported as follows:

Error in Preambles: cannot understand the alias declaration:

alias(,a@research.rutgers)

– Cannot successfully prepare the law, due to: preamble failure

Please input the law file name with path (or URL):

2.1.3 Syntax validation and law loading

After the preamble, the body of the law is validated. This step is language dependent. In the case of Java laws, the body of the law is compiled; in the case of Prolog laws, a Prolog engine consults the law body. During this step, if any syntax error is encountered, it is reported along with its location. Due to the different characteristics of the interpreter, there is slight difference between Java and Prolog laws: In the case of Java laws, whenever an error is reported, the tester demands for the user to correct the law and reload it:

Cannot successfully load the law, due to:

compilation failure

Please input the law file name with path (or URL):

In the case of Prolog, the laws can still be loaded even in the presence of errors (due to the nature of the interpreter, certain rules can be evaluated, independent of the presence of other faulty rules). An example of output for a faulty Prolog law follows.:

– Do you want to accept this law?

– Press enter or "yes" to accept, "no" to try another law:

no

Please input the law file name with path (or URL):

When a syntactically correct law is loaded, the following message is displayed:

– Successfully loaded the law

Note that prior to loading it, a Prolog laws is transformed for testing purpose (see the Moses manual for details of this). Therefore the provided law differs from the law that is loaded. Whenever a rule is reported as faulty, the transformed law is displayed. However, the correspondence is intuitive.

2.1.4 Configuration loading

After a law is loaded, various events can be submitted to evaluation. This evaluation, however is performed in a certain context. This context can be defined by specifying the following environment variables: *CS*, *DCS*, *Peer*, *PeerHash*, *RList*, *AList*, *Self*, *ThisLawName*, *ThisLawHash*. These variables are taken from a file as follows:

*Please input the configuration file name with path:
(Press enter to skip this step, setting the context to its default value)
ConfigurationFileName*

Each entry in the file is a term context(name,value). An example of content of a configuration file is:

```
context(Self,a@mco.rutgers.edu)
context(CS,[budget(budget(3)),position(client),ticket(1),time(100)])
context(ThisLawName,testlaw)
context(PeerHash,122222222)
context(Peer,peer@domain.org)
```

The configuration file can be updated and reloaded at any time during the testing session. If no configuration file is specified, the system will automatically set the context variables to their default values. Certain variables in the context cannot be manually set: *Msg*, *SubjectHash* and *Clock*. Variable *Clock* is adjusted based on the system current time, while the others are event dependent.

2.1.5 Dynamic context updating

Beside loading the entire configuration file, the context variables, including the control states, can be set during the testing, dynamically. The following commands are used for this:

context(VariableName, VariableValue): sets the variable "VariableName" to the value "VariableValue". The variable name could be one of: *CS*, *DCS*, *Peer*, *PeerHash*, *RList*, *AList*, *Self*, *ThisLawName*, *ThisLawHash*;

New Event: (!h for help):context(Self,lgi@rutgers.edu)

The context has been updated, !c to check the updated context re-asserted the context of the law;

New Event: (!h for help):context(CS,[budget(1)])

The context has been updated, !c to check the updated context re-asserted the context of the law;

add(NewTerm): specially designed to manipulate the control state CS: it adds the term "NewTerm" to the current control state;

*New Event: (!h for help):add(lastmessage(hello))
doAdd(lastmessage(hello))
re-asserted the context of the law
The context has been updated, !c to check the updated context*

remove(OldTerm): specifically designed to manipulate the control state CS: remove the term "OldTerm" from current control state;

*New Event: (!h for help):remove(role(vendor))
doRemove(role(vendor))
re-asserted the context of the law
The context has been updated, !c to check the updated context*

!c: in order to view the context of the law;

*New Event: (!h for help):!c
The context of the law is:
CS = [budget(1),lastmessage(hello)]
DCS = []
Msg = null
Peer = peer@rutgers.edu
PeerHash = 222222222
RList = []
AList = []
Self = lgi@rutgers.edu
SubjectHash = null
ThisLawName = thisLawName
ThisLawHash = 111111111
Clock = 1121108476622
re-asserted the context of the law
prolog context is: context([budget(1),lastmessage(hello)],[],'peer@rutgers.edu',
222222222,[],[],'lgi@rutgers.edu',thisLawName,111111111,time(311419,76622))*

!r: it reloads the configuration file, if such a file has been previously loaded;

2.2 Event evaluation

In order to discover runtime errors, a user should be able to interact with the tester by submitting events at the command line:

New Event: (!h for help):

The events are defined by their names and their parameters. After an event is evaluated in a given context, a list of primitive operations is returned.

2.2.1 Java laws

In order to test the event `sent(String source, String msg, String dest, String law)`, the user has to provide a text as below. The result of the event evaluation is provided:

*Evaluate the event: `sent(x@aol.com,somemessage,y@yahoo.com,thislaw)`
With the control state `[]`
The operations are: `doForward()`
The new control state is : `[]`*

By examining the operations returned, the user can determine whether the evaluation has performed according to the desired model. If some of the resulting primitive operations are intended to affect the control state, this is updated accordingly:

*New Event: (!h for help):`sent(source,addstring(budget(1)),dest,law)`
Evaluate the event: `sent(source,addstring(budget(1)),dest,law)`
With the control state `[]`
The operations are: `doAdd(budget(1))`
The new control state is : `[budget(1)]`*

When the user provides an event that does not match any of the LGI events, the tester will provide the following message to the user:

*New Event: (!h for help):`sent(x@aol.com,somemessage,y@yahoo.com)`
Evaluate the event: `sent(x@aol.com,somemessage,y@yahoo.com)`
With the control state `[]`
The operations are:
Error: *The parameters you submitted are not correct, please follow the API:*
`sent(java.lang.String,java.lang.String,java.lang.String,java.lang.String)`*

2.2.2 Prolog laws

Whenever a valid event is presented, a Prolog law returns a list of primitive operations, as in the case of Java laws. If this ruling contains invalid operations, or operations with wrong syntax, the tester will notify the user by placing these operations in a separate category. The operations regarding the control state are executed, and the CS is updated, as in the case of Java. If the following law is tested:

*`adopted(Par,Cert) :- do(deliver(law,adopted,Self)), do(add(budget(0))),
do(add(visits(0))),
do(delive(law,adopted2,Self)),`*

do(forward(law,Self)).

If the following event is tested:

New Event: (!h for help):adopted(Par, Cert).

Evaluate the event: adopted(Par, Cert).

With the control state []

*Valid operations: [do(deliver(law,adopted,'lgi@rutgers.edu')),
do(add(budget(0))), do(add(visits(0)))]*

*Invalid operations: [do(delive(law,adopted2,'lgi@rutgers.edu')),
do(forward(law,'lgi@rutgers.edu'))]*

Updating the Control State... doAdd(budget(0)) doAdd(visits(0))

*The new control state is : [budget(0),visits(0)] re-asserted the
context of the law*

You may notice in the above output that the operations are categorized into two types. Some of them are valid operations *do(deliver(law,adopted,'lgi@rutgers.edu'))*, *do(add(budget(0)))*, *do(add(visits(0)))*, obtained from the ruling of the law. Note that 'lgi@rutgers.edu' comes from the context variable "Self" defined previously; also note that the control state is updated accordingly. There are also invalid operations, most of the time they are operations with syntax errors: *delive(law, adopted2,'lgi@rutgers.edu')* (misspelling) and *forward(law, 'lgi@rutgers.edu')* (invalid number of parameters). The user will thus be able to determine the validity of the evaluation, and to discover the wrong operations.

2.2.3 Other commands

A number of other commands are provided in order to facilitate the testing and debugging. They are shown as follows:

!h: help command;

!s: view the source code of the law;

New Event: (!h for help):!s

The source code of the law is:

law(L2,language(prolog)).

sent(X,ping(M),Y) :- not(pingTo(Y)@CS), do(add(pingTo(Y))), do(forward).

arrived(X,ping(M),Y) :- do(add(pingFrom(X))), do(deliver).

sent(X,pong(M),Y) :- pingFrom(Y)@CS, do(remove(pingFrom(Y))), do(forward).

arrived(X,pong(M),Y) :- do(remove(pingTo(X))), do(deliver).

disconnected :- do(quit).

!l: view the transformed (also called massaged) code of the law, i.e. the law executed by the testing interpreter; this is useful in order to retrieve the mapping between the submitted law and the actual tested law.

!v: view all the regulated events of the law;

New Event: (!h for help):!v

The regulated events of the law are:

sent(X,ping(M),Y)
arrived(X,ping(M),Y)
sent(X,pong(M),Y)
arrived(X,pong(M),Y)
disconnected

!q: quit the tester;

3 Implementation of the law tester

3.1 Work flow of the system

This section provides some implementation details related to the tester. The tester code for Java and Prolog laws are based on the same module, implemented in the class `moses.testers.Tester`. The work flow of this module is shown in the following figure.

Due to the inherent differences between the Java and Prolog languages, there are differences in the implementation of Java and Prolog laws. As observed in the work flow, the main difference is in the event evaluation process, and in the pre-processing/conditioning of the law text. The other modules, including those to handle extra commands, are independent from the language.

The next two sections outline the event evaluation, for Java and Prolog laws, respectively.

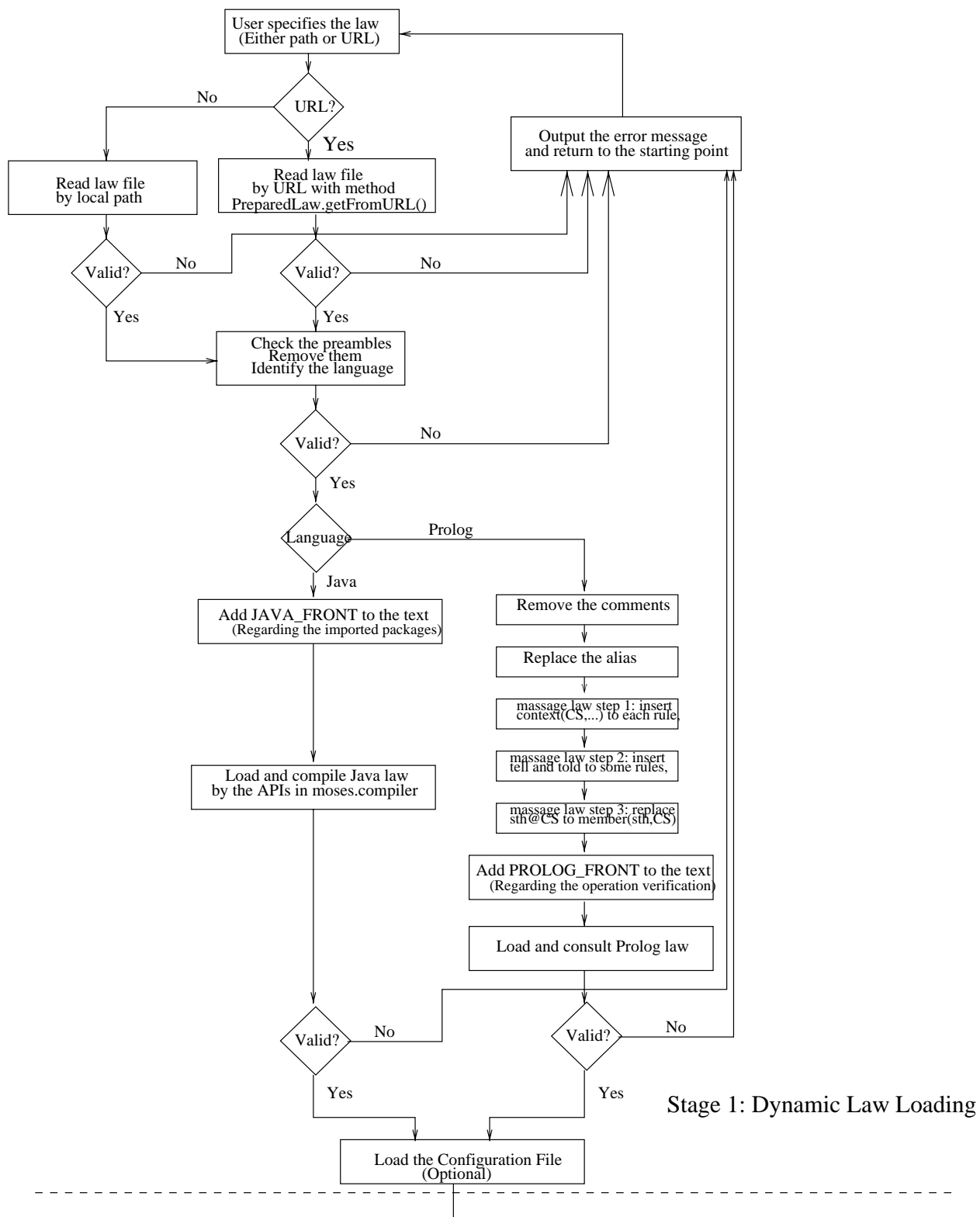


Figure 1: The work flow of law tester

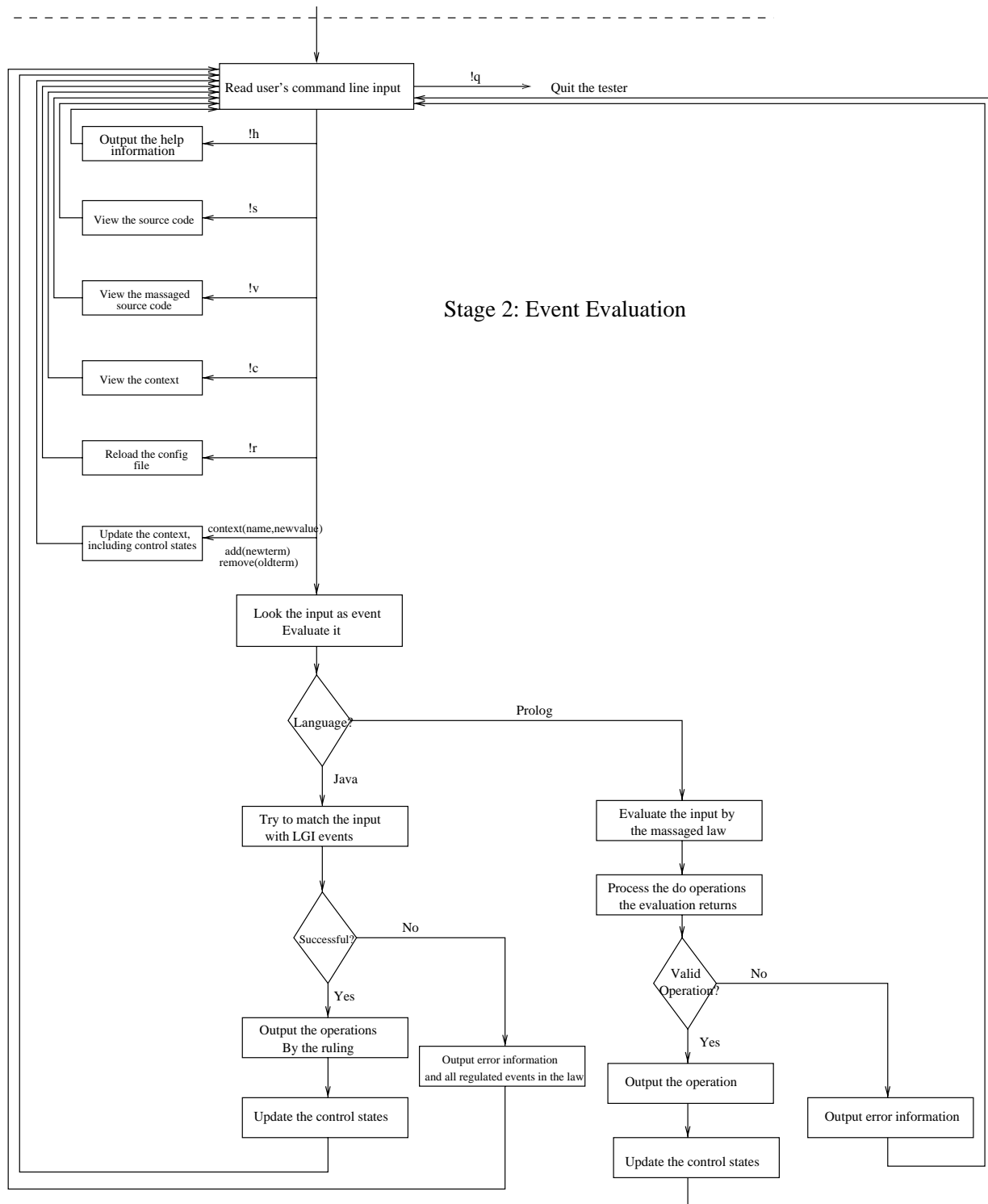


Figure 2: The work flow of law tester (Cont.)

3.2 The event evaluation for Java laws

In the case of Java laws, every law class extends the `moses.controller.Law` base class. All the events and primitive operations are defined as methods part of this class. As a result of invoking an event method, certain primitive operations will be called based on the law implementation. Take the example of the following law L0:

```
law(L0,language(java))
public class L0 extends Law {
...
    public void sent(String source, String message,
                    String dest, String destlaw) {
        doAdd(message);
    }
...
}
```

Observe that law “L0” extends the class `moses.controller.Law` (here denoted simply as `Law`). When the event `sent(String source, String message, String dest, String destlaw)` is invoked, the method `doAdd(message)`, corresponding to a primitive operation, will be called. In the base class `moses.controller.Law`, this method is defined as follows:

```
public class Law {
...
    public void doAdd(String st) {
        DoOperation doop = new DoOperation();
        ...
        doList.add(doop);
    }
...
}
```

Thus, the method adds an operation instance to the ruling of the event (a Do-List which contains all the operations to be processed).

The tester is based on the same idea. A law loaded in the tester extends the class `moses.testers.Law` instead of the standard `moses.controller.Law`. This process is transparent to the user, however, because the importing clause resolving the `Law` package is added during the dynamic loading process. In the tester `Law` class, the primitive operations are implemented in order to display the operations on the screen, and to update the control state, immediately, if necessary. For instance, operation `doAdd` is defined as:

```
public class Law {
...
    public void doAdd(String st) {
        controlState.csList = CS;
        controlState.add(Term.parse(st), at, RList, AList);
        System.out.println("doAdd(" + st + ")");
    }
...
}
```

Consider the case when the user would like to evaluate the event: *sent(source, budget(1), dest, law)*. The first step performed by the tester is to try to unify the input with a valid event syntax. In this case, the event will be successfully unified with the method *L0.sent(source, budget(1), dest, law)*. As defined previously, the law will execute the operation *doAdd(budget(1))*. Given that our law extends the tester Law class, the operation *doAdd(budget(1))* will have no effect beside printing the information on the screen and adding the term *budget(1)* to the control state.

Thus the user is able to observe what primitive operations are executed and to verify the ruling of the law for a given event *sent(source, budget(1), dest, law)* and with a given control state, for testing and debugging purpose.

3.3 The event evaluation for Prolog law

Prior to loading and evaluating a Prolog law, the law has to suffer some transformation (also called massaging).

Let us take the example of this simple law with only one rule:

```
law(L0,language(prolog)).
sent(X,M,Y) :- do(forward), do(add(M)), do(somethingwrong).
```

After the massaging process, the law will contain, among others, the following rules:

```
...
do(forward) :- println(0),println(do(forward)).
...
do(add(T)) :- println(1),println(do(add(T))).
...
do(Op) :- println(2),println(do(Op)).
...
sent(X,M,Y) :- tell('25187do.pl'),
context(CS,DCS,Peer,PeerHash,RList,AList,Self,ThisLawName,ThisLawHash,
Clock),
do(forward), do(add(M)), do(somethingwrong), told.
```

The change is obvious: the original rule is enhanced into a more complex one, and a number of additional rules are also inserted in the law. To be more specific, whenever a Prolog law is loaded, it is massaged in the following ways:

```
context(CS,DCS,Peer,PeerHash,RList,AList,Self,ThisLawName,ThisLawHash,
Clock) is inserted in each rule body. This is necessary in order to provide
the dynamic context variables along with the event, during the Prolog
evaluation.
```

A number of rules are added at the beginning of the law; these rules are static. They all offer an implementation of the primitive operations, like *do(forward) :- println(0),println(do(forward))*.

Firstly, when a primitive operation is invoked, the corresponding rule will print the operation term to a temporary file. Additionally, these rules pro-

vide a means to verify the syntax of the primitive operations, categorizing them into three types, with code “0”, “1” and “2”.

The tester processes the primitive operations by reading this file, and identifies the category of the operations by their codes: “0” denotes a valid operation, with no control state modification, “1” stands for a valid operation with control state modification, and “2” stands for an invalid operation, i.e. the operations with syntax error.

A pair of terms *tell(FileName)* and *told* are inserted at the beginning and at the end of each rule. They are to specify this temporary file name where the primitive operations are stored. The tester will remove the temporary file after the entire evaluation is completed.

In the above example, whenever the user tests the event *sent(x,budget(1),y)*, the tester will evaluate it using the massaged law presented above. The law computes the ruling by writing to the temporary file “25187.pl” :

```
0
do(forward)
1
do(add(budget(1)))
2
do(somethingwrong)
```

The tester subsequently reads the temporary file “25187.pl” (implemented as a “pipe”). It retrieves the three primitive operations, and it handles them separately by category codes:

do(forward): valid operation, outputs it to the user.

do(add(budget(1))): valid operation, outputs it to the user, meanwhile updates the control state, by adding term *budget(1)* to it.

do(somethingwrong): invalid operation, outputs it to the user along with an error information in order to notify the user.

This way the user is able to observe the output, and to verify the ruling of the law, under a given event *sent(x,budget(1),y)*, and with a given control state), for the testing and debugging purpose.