1 Supporting Members of an Open Community

In an open distributed system, what a component may do or may not do, along with the consequences of such actions, may change dynamically, as the entire system in question functions. Thus, in order for components to operate effectively in such a dynamic environment, they need to be supplied, in a timely manner, with essential information that characterizes the circumstances in which they operate. Among such information, we consider in particular what would be derived as answers to the following kinds of queries:

- “What can this member do? What are the alternatives?”
- “What happens if a certain event occurs to this member? How about a certain series of events?”

The above kinds of information would tell how the entire system, which a member in question participates in, currently circumscribes the member’s immediate action and the consequences of (hypothetical) series of actions of itself and of others. Based on such knowledge, the member would be able to make an informed decision as to its next steps, while operating within the system. For example, in the buying team of Chapter [sec-wde], having just received an assignment, a buyer may learn that he cannot issue a purchase order until he gets some budget, without gleaning the fact from more complex knowledge on how the entire system functions—in this case law $BT$.

In this chapter, we pursue the theme of providing members with the above kinds of essential information. In particular, we investigate how such functionality can be implemented to serve members of a community governed by a certain law under LGI. And we demonstrate the functionality by presenting a user interface that utilizes such functionality for human members.

At this point, we introduce the terminology used in the rest of this chapter. The first kind of queries mentioned above translates in the context of the LGI model as follows:

Given the current (“real”) control-state, derive all events (of a certain type) that would yield non-empty ruling.

We henceforth call this kind of queries as possible-event queries. Similarly the second kind of queries mentioned above translates as follows:

---

*Adapted from the chapter titled “Operating within a Regulated Community” – part of the Ph.D. thesis of Takahiro Murata, October 2003. This work has been completed while the author was affiliated with Rutgers University.
Given a hypothetical control-state (which could be a “copy” of the real one) and a specified regulated event, derive the corresponding ruling and, if it is non-empty, the resulting control-state.

We henceforth call this kind of queries as testing queries. To the best of our knowledge, the very fact that given the above mentioned motivation, these kinds of simple, yet explicit queries are possible on a deployed system—as we show in this chapter—is unique to LGI.

The rest of this chapter is organized as follows: In the next section, we provide an overview of the user interface, describing concisely its three main features: connection, operation, and testing, while pointing out that the use of these features is regulated by the law of the community the user joins. In Section 3, we explain the connection feature, i.e., how a user can establish a connection with an LGI controller, and become a member of a certain community. Our user interface provides the above kinds of essential information in a timely and flexible manner; the implementation of such functionality is explained in two sections: in Section 4, focusing on the treatment of the first kind, derived from possible-event queries, and in Section 5, on that of the second kind, derived from testing queries. Throughout, we describe not only the operations but also the rationale behind certain designs, with which we hope to convey the value of this mechanism, especially for agents operating within an open community.

2 Overview of the Interface

Our user interface—an operational gateway for humans acting within a community governed by a law under LGI—provides three main features:

(a) connecting to a controller under a specified law;

(b) operating within a community; and

(c) local testing of communal interaction.

We explain briefly all these features, in the above order.

First, a user wishing to join an $\mathcal{L}$-community must establish a connection with a controller. The user may know of the identity of the controller (i.e., the IP host and the port number); or, otherwise, he/she\(^1\) can contact a controller-server through the interface for a set of controllers known to it. In either case, the interface allows the user to contact the controller of his choice, while specifying the law to be adopted and his name to be used in the community. Assuming that the text of the specified law is available, and is well-formed, while no other agent has been using the specified name, a connection between the interface and the controller is established, and the user starts operating within the community, through the interface.

Second, once the user has joined the community, the interface allows him to dispatch towards the controller messages addressed to specified recipients, and to receive those delivered to him by the controller. Moreover, by utilizing the mechanism to process possible-event queries (Section 1), the interface shows, in a list called repertoire, all the kinds of regulated messages that the user can send at the moment within the community. Thus, in composing a regulated message to send, all the user needs to do is to select an item from the repertoire, which works as a “template” of the message, and to fill out the parameters in it. This way, the user is provided, for his disposition, the first kind of essential information discussed in Section 1.

\(^1\)Henceforth in this chapter, to be concise, we only use “he” as the singular pronoun of the third person.
1. \texttt{adopted(Args)}
   :- (humanUser@Args ->
       do(show(possibleEvent([certified,sent]),
       cs([manager,buyer,'asmt/4','budget/1','mgr/1'])),
       do(imposeStateObligation([manager,asmt,budget])), do(enterTest))
     ; true).

   If a \texttt{humanUser} atom is supplied in the arguments, a set of primitive operations are executed; otherwise the ruling is empty.

2. \texttt{stateObligationDue} :- do(show(possibleEvent([certified,sent]),
    cs([manager,buyer,'asmt/4','budget/1','mgr/1'])),
    do(imposeStateObligation([manager,asmt,budget])).

   Whenever any change occurs to the terms on which the state-obligation is imposed, this rule is triggered.

3. \texttt{sent(X, show(Es,Ts), Y)} :- do(show(possibleEvent(Es),cs(Ts))).

   The user can request the possible-event query to be executed when he needs the answer.

4. \texttt{sent(X, exitTest, Y)} :- do(exitTest).

   The user can exit the testing mode.

Figure 1: A fragment of a law to regulate the interface

Finally, while the user is operating within the community, the interface allows him to explore hypothetical transitions between states, by utilizing series of testing queries (Section 1). This functionality, which we call \textit{local testing}, provides the second kind of essential information discussed in Section 1. The term “local” is meant to capture the fact that these transitions are carried out by this particular user alone, solely based on his local control-state (along with the constraints imposed upon him by the law). Note therefore that the feature of “communal” testing, where multiple users may be involved in conducting hypothetical, not necessarily local, state transitions, is beyond our scope at this time.

Two technical remarks: First, the use of the features described above is regulated, to various degrees, by the specific law the user adopts. Such regulation includes in particular how often or at which point the repertoire is updated, and when or on which conditions the user is allowed to use the local testing feature. Figure 1 shows a sample fragment of a law that contains rules to achieve such regulation, which we will refer to, while presenting each feature of the interface individually below.

Second, we have two kinds of implementations of our user interface: one as a Java applet and the other as so called a Java Swing application, which we henceforth call the \textit{applet version} and the \textit{application version}, respectively. This division is mainly because of the standard access-control placed on applets by the commonly available web browsers (e.g., Netscape 4.x); running in such a web browser, applets are not allowed to access, particularly, the local file system where we assume the user keeps \textit{certificates}. The applet version does not attempt to override such constraints, providing no means regarding certificates. In contrast, in the absence of such restrictions, the application version allows the user to present certificates, stored in the local file system.
3 Connection

We discuss steps required for the user, operating through the interface, to establish a connection with an LGI controller, i.e., to join an LGI-community under a chosen law. We first explain the steps with the applet version of the interface, and then mention the difference that arises in using the application version.

There are two ways to start the user interface as follows. If the user knows of (a running instance of) the LGI controller-server, he can visit its web page via a web browser. For example, assuming that the controller-server is running on paul.rutgers.edu, and listening to port 9020, the URL to be specified is as follows:

http://paul.rutgers.edu:9020/sever.html

This controller-server’s web page provides hypertext links to all the controllers known to it. The user can simply click one of them that represents the controller to connect to; this way, by accessing the controller’s web page, the applet version of the interface (is downloaded and) starts running in the user’s web browser. If on the other hand the user knows of the controller that he wants to connect to, he can directly visit the controller’s web page. For example, assuming that a controller is running on paul.rutgers.edu, listening to port 10000 for new clients, which sets up its HTTP port at 9500 (i.e., 500 less the TCP port) in the current implementation, the URL to be specified is as follows:

http://paul.rutgers.edu:9500/ControllerInterface.html

Once the controller is chosen, and the applet is running, the next step is to adopt a law, i.e., to establish a connection with the controller as a member of the community governed by the law. The applet has the initial display like the one in Figure 2. In the figure, notice that the location field of the browser shows the URL of the controller’s web page. In this initial window of the applet, the user is supposed to supply the relevant information in each text-field up to just above the Enter button. The input to the two text-fields from the top is mandatory; i.e., (a) the URL of the law the user intends to adopt; and (b) the (local) name of the user, which, together with the host-name of the controller, constitutes his id in the community.

The user should specify the mailbox password, which will be used when reconnecting to the controller, after previous disconnection, to retrieve the messages accumulated in his mailbox maintained by the controller. Thus, the user should select the radio button named Reconnect, below the text-field for the mailbox password, only when he is reconnecting, in which case, the mailbox password must match the one specified previously.

Pressing the Enter button causes an attempt to adopt the specified law at the target controller. Note that the input to a text-field, titled Adopting Args, just above the Enter button, is optional; but if present, this input forms the argument list in the adopted event that will be generated as the first regulated event when the user’s attempt to adopt the law is successful. E.g., in Figure 2, a term humanUser is entered, which would be processed by rule R1 of Figure 1 (assuming this fragment is part of the law the user specifies), leading to the execution of a number of primitive operations. (These operations are central to the functionality described in this chapter, and will be explained in Sections 4 and 5.)

Finally, if any error occurs in the above attempt (e.g., the specified name is already used by some other agent, or the specified law is not available or malformed), it is reported next to the Enter button (and naturally unless the cause of the error is rectified, the law adoption will not succeed).

---

2 If several terms are to be given in the Adopting Args field, they must be separated via a comma.
3.1 The difference in the application version

We now explain the steps of connecting to a controller, when using the application version, focusing on the difference from those with the applet version. First, to start the application version of the interface, the following command should be entered from the shell:

```prompt> java moses.ui.UserConsole ['controller-server host', 'controller-server port']```

The use of Java 1.3 or higher is recommended. In the above command line, `UserConsole` is the name of the Java application given to the user interface, and we assume that the `CLASSPATH` is properly set so that the corresponding class files can be located as installed on the machine being used. There are two optional arguments, for the host name where the controller-server is running, and for its port number.

Initially, the user sees a dialog window that looks like the one in Figure 3. If the controller-server is specified in the above command line, the dialog window lists the information on all controllers currently known to the controller-server (in the figure, the one on `miriam.rutgers.edu` at port 10000 is only known). The user can select at most one of them as the controller to connect to, by selecting the radio button beside the information of a controller. If the user wants to connect to a controller not known to the controller-server, or if no controller-server is specified in the above command line, he can enter the host name and the port number in the text-fields located at the bottom of the window. The user can then press the `go` button to move to the next dialog. (Pressing
the clear button clears away the selection and the text entered in the text-fields.

The next dialog window looks like the one in Figure 4. The dialog indicates, at its top, the controller an adoption attempt is to be made with. Most of the text-fields and the buttons along with their meaning are parallel to those of the applet version, and thus not be repeated here. However, this window has the following additional feature: If the user has doubts as to what to enter in the adopting args text-field, he can consult the relevant rules that process the adopted event in the specified law. To do so, he should press a button titled adopting rules?, located towards the bottom of the window, on the left, which causes these rules with comments (if any in the law) to be displayed in the text-area to the right of that button. (The applet version does not provide this kind of query, due to the standard restriction on the applet’s ability to open a network connection to a third host; instead the law server’s web interface should be consulted.)

Finally, note that here is a difference between the two versions in the connectivity between the interface and the controller: In the application version, the above dialog (Figure 4) is solely given by the interface, irrespective to the actual availability of the target controller. However, in the applet version, by the time the initial window appears as seen above (Figure 2), the HTTP connection has been established between the web browser and the controller. Hence, the subtle difference in the names of the action buttons to establish a connection with the controller as a member of the community: Connect in the application, whereas Enter in the applet.

4 Operation

Shown in Figure 5 is a snapshot of the console window of the applet version of the interface. This console window appears when the connection to the controller has been established, and the user has joined a community under the specified law, as described in Section 3. We proceed with presenting operational features of our interface, by referring to the applet version, and supplement it with differences in the application version later.

For the sake of presentation, we divide the operational features into two categories: one consisting of those that are basic for the user to exchange messages with other members in the community,
and the other of those to assist the user, utilizing possible-event queries (Section 1), which we call repertoire.

**Basic features:** There are several features in this category as follows:

1. At the top center of the window, the information regarding the member status is shown, consisting of the member's name, local to the controller, the controller's host and port, and the URL of the law.

2. In the *message* text-field, the user can enter a message to dispatch, while specifying an intended recipient in the *recipient* text-field. Pressing the *send* button dispatches the message addressed
to the recipient towards the controller. (Thus, when the message is received by the controller, it causes a \texttt{sent} regulated-event.)

(3) Pressing the button marked \textit{peers}, just next to the \textit{recipient} field, causes a separate dialog window to open up, containing ids of all members of the community that the user has communicated with so far. Clicking one of the items in the list pastes it in the \textit{recipient} field, possibly saving repeated key-typing in sending messages.

(4) In the large text-area towards the bottom of the window, the user can view, the messages that he has dispatched, and those that have been delivered to him, in the chronological order.

(5) The button at the top-left corner of the window, currently marked \textit{disconnect} in the figure, allows the user to temporarily disconnect from the controller. When disconnected, the text on the button changes to \textit{reconnect}, pressing which re-establishes the connection with the controller. (Although in general reconnecting to a controller requires the matching mailbox password, for this reconnection there is no need of explicitly supplying it, which is maintained by the interface.)

Since the repertoire feature is more involved, we dedicate a section below.

4.1 Repertoire

Our user interface provides a \textit{complete} set of regulated messages that the user can send at the time; complete, in that no other kinds of messages would result in non-empty ruling (i.e., causing certain effects), given the control-state used in the possible-event query to derive this set of information—hence, the set is called \textit{repertoire}. Assisted with the repertoire, in composing a message to dispatch, instead of figuring one out from scratch, the user can simply choose one, while making sure that he is giving consideration to all kinds of messages that would cause an effect at the time.

The repertoire appears as a list titled \textit{repertoire} around the middle of the applet window (Figure 5), on the right. One direct way to use the repertoire is to examine the current choices of the messages to send. In addition, by selecting any one of the items in the \textit{repertoire} list, the user can have it pasted to the \textit{message} field, where he can further edit the parameters, if applicable, before dispatching it addressed to some recipient. Moreover, when an item is selected in the \textit{repertoire} list, the rules in the law that would yield non-empty ruling upon sending the selected item are shown, along with the comments if any, in the text-area titled \textit{relevant rule} to the right of the \textit{repertoire} list. (We will explain the use of the text-area titled \textit{CS} later.)

Note that there are two factors that introduce imprecision to the repertoire, as follows. First, although, as we will show below, we provide a number of ways to control the point at which the repertoire is updated, we do not explicitly synchronize the update of the control-state (in the controller) with that of the repertoire (in the user interface)—let alone with the user’s dispatching a message towards the controller—to keep the controller’s implementation lean and clean. Thus, it might happen that the user dispatches one of the messages in the repertoire, only to find it causing no effect due to the control-state having changed in the meantime (because of, e.g., the arrival of a message from another member). Second, in computing the answer to a possible-event query, when arithmetic is involved in some rule, only conservative approximation is possible. For example, consider the following rule in a law:

\begin{verbatim}
sent(X,m(A),Y) :- budget(B)@CS, B>=A, do(forward), do(decr(budget(B),A)).
\end{verbatim}
In this rule, without having the binding of \( A \), even if term \( \text{budget}(B) \) exists with a ground binding on \( B \) in the control-state, it is not precisely determined whether sending of \( \text{m}(A) \) results in non-empty ruling, due to \( B \geq A \) not being determinate. But this \textit{sent} event is included in the answer to the possible-event query, to be conservative.

Notice that in the applet window, there appears no widget, by operating which the user could directly cause some effect to the repertoire; any such effect is actually achieved under intricate regulation by the law, as we explain next.

**Control by the law over the repertoire:** The structure of a member in an LGI-community can be divided into two parts: the controller serving the member, and the client program that communicates with the controller. When making the distinction between these two clear, we call the latter the \textit{actor}. In a case where the actor has no capability of utilizing the repertoire information, providing such information would simply be a waste, or even worse, it might break the actor program. Note also that the information that constitutes the repertoire is directly related to the control-state of the agent in question, which may need to be kept undisclosed in some cases. Therefore, in general, communal regulation LGI provides over the interaction of members should also apply to which members get the repertoire information when or under which conditions.

Such control is made possible via a primitive operation, \textit{show}, which implements the possible-event query. (We will explain the detail of this primitive operation below.) By stipulating the conditions under which this operation is executed, the law can provide for various control as to when to invoke the possible-event query. Among such control, two modes are exemplified in rules in Figure 1 (in page 3) as follows:

(a) The law can allow the user to invoke the query at the time of his choice, as seen in rule \( \mathcal{R}3 \), by which the user can send a \textit{show} message, to get the result of the corresponding possible-event query.

(b) Notice that unless the control-state changes, the answer to a possible-event query remains the same. Thus, one can automate the triggering of query invocation by having provisions to “sense” the change in the control-state. In fact, this sensing is provided by the LGI \textit{state-obligation}, and rules \( \mathcal{R}1 \) and \( \mathcal{R}2 \) utilize it to watch over any changes made on terms, \textit{manager}, \textit{asmt}, and \textit{budget}. Specifically, when the user joins the community, \( \mathcal{R}1 \) is triggered, and consequently, the \textit{imposeStateObligation} operation is executed, specifying the above terms to be watched. (We assume that the user has provided a term \textit{humanUser} as one of the arguments to the \textit{adopted} event, which is a typical example of how the law can distinguish an actor that would benefit from the kind of query processing discussed in this chapter, and one that would not.) Thereafter, whenever a change occurs to any one of those terms, a \textit{stateObligationDue} event is generated. This event is processed by \( \mathcal{R}2 \), which executes the \textit{show} operation (and re-imposes the state-obligation). Combined with the execution of the \textit{show} operation in \( \mathcal{R}1 \), the user (automatically) gets updated repertoire information immediately after joining the community, and thereafter whenever any change occurs to the above three terms.

Figure 6 depicts the above kinds of interaction between the interface and the controller.

**Primitive operation \textit{show}:** Fundamental to support the manipulation of the repertoire is a \textit{primitive operation} \textit{show}, which has two parameters, \textit{possibleEvent} and \textit{cs}, as follows:

\[
\text{show(possibleEvent(Etypes),cs(CTerms))}
\]
When executed, this operation discloses two kinds of information in a single payload to the actor as follows: First, it discloses the answer to the possible-event query for the regulated events given in the list bound to \texttt{Etypes}. For example, by rule $\mathcal{R}1$ of Figure 1, the query is made about two types of regulated events: \texttt{certified} and \texttt{sent}. The evaluation of this query is done based on the control-state that is reached at the end of reflecting all changes required by the current ruling (i.e., the ruling that includes this \texttt{show} operation).

Second, if terms given in the list binding of \texttt{CStems} exist in the control-state as mentioned above, they are disclosed to the actor. For example, given the binding seen in rule $\mathcal{R}1$, if a term (of any arity) whose functor is \texttt{manager} exists in the control-state (thus, including an atom \texttt{manager}), it is disclosed. If the arity of terms in the control-state needs to be distinguished, one can use a particular syntax also shown in $\mathcal{R}1$; e.g., \texttt{`asmt/4`} allows only the disclosure of a term whose functor is \texttt{asmt} with four arguments. (Note that, in the rule, such a term specification needs to be enclosed in a pair of single-quote ('').)

Upon receiving the output of the \texttt{show} operation, the user interface updates, as seen in Figure 5, the content of the \texttt{repertoire} list. (For the message format of this output, see Appendix A.) The terms disclosed from the control-state appears in the \texttt{CS} text-area. The information regarding possible \texttt{certified} events is also utilized as will be shown below, when we discuss the handling of certificates in the application version.

### 4.2 The difference in the application version

When using the application version, the user sees the main console that should look like the one in Figure 7 show up. The window has menu buttons on its left side, and by pressing them, the user can open up corresponding sub-windows, or utilize the associated functionality.

The main difference between the applet version and the application version is (the lack or availability of) the handling of digital certificates although they differ somewhat in their look and feel. We outline the feature associated with each menu button, and then explain how certificates are handled.

The top button, labeled \texttt{Connection}, opens up a window that displays the member information comparable to that shown in the top center portion of the applet window. The second button, labeled \texttt{Send certificate}, allows the user to present a certificate to the controller; the detail of such operation is explained below. The third button, labeled \texttt{Compose Message}, opens up a sub-window, with which the user can conduct operations comparable to those available in the applet version, including other types of events, e.g., \texttt{arrived}, in the binding of \texttt{Etypes} of the \texttt{show} operation would result in no additional effect on the user interface, although the information regarding such types of events is disclosed.
except viewing messages dispatched and those delivered, which is available in the sub-window that opens when the fourth button, labeled In/Out Messages, is pressed. The functionality associated with the fifth button, labeled Disconnect, is the same as that associated with the disconnect button of the applet.

**Presenting a certificate:** A dialog that looks like the one in Figure 8 opens when the user presses the Send Certificate button of the (application) console. Note that this dialog window is modal; i.e., once it is opened, until this window is closed (or disposed), the user is not able to operate on the other parts of the user interface. This design decision is made, considering the sensitivity of this particular operation in the light of security.

In the top text-field of the dialog, labeled Certificate, the user should enter the name of the certificate file (including the path either absolute or relative to the directory where the user interface has been invoked). If this is meant to be a self certificate, the user should also enter, in the next text-field, labeled Signature, the name of the signature file that has been created by signing the certificate (by his private key) as given above.

When the Send button is pressed, if the specified file(s) is not found, the dialog will remain open, without any error warning (at this time). On the other hand, if the file exists, and successfully sent out toward the controller, the dialog window closes.

A text-area below the Send button, labeled possible cert’s shows the kinds of certificates the law accepts from the user (at this time). The disclosure of such information is done by the controller when the corresponding show operation is executed, containing certified in its possibleEvent parameter, as seen in R1 of Figure 1. The information in this text-area consists of the following: (a) the pattern of each certificate, similar to an item in the repertoire; (b) the rule that will be
triggered when such a certificate is presented; and (c) any comments associated with the rule in the law. These help the user recognize which certificate he should submit.

We did not discuss the functionality associated with the testing button of the applet or the sixth (bottom) menu button, labeled Testing, of the application, which will be the topic of the next section.

5 Testing

Shown in Figure 9 is a snapshot of a sub-window, titled testing, that shows up when the user presses the testing menu button of the application version of the interface. In the applet version, a similar (separate) window pops up when the user presses the testing button located in the top-right corner of the applet window. Since the functionality provided through these windows, one from the application version and the other from the applet version, is the same, henceforth we refer to either of them as the testing window indiscriminately. We also call features provided through the testing window collectively as the testing feature.

Based on the testing query (Section 1), available through the testing window, the user can explore hypothetical state transitions, by specifying the control-state to be used for the ruling, while checking if the ruling is non-empty, and if it is, resulting in what kind of new control-state and associated primitive operations. Repeating such experiments allows the user to view a sequence of hypothetical state transitions. This way, members of the community can be better informed of the circumstances they are operating in, while possibly making them be more familiar with the law of the community.

In addition, the testing feature can be used to “debug” a law since it allows the user to compare the state transitions he expects and what would happen according to the law as it is written. Note that in contrast to the possible-event queries discussed in Section 4, testing queries are precise, in that the ruling can be determined unambiguously.

The following two sub-features form the testing feature:

(a) querying all regulated events that could happen, which is akin to the possible-event query explained in Section 4; and

(b) issuing a testing query, i.e., the test-evaluation on any type of regulated event.

Note that in using either feature, the user can choose the control-state on which the intended query is based between (a copy of) the real one and a hypothetical one. We explain these features below,
which are implemented by utilizing controller’s *testing mode*. (For the programming interface to invoke features provided by the testing mode of the controller, see Appendix B.)

**Feature (a):** The user can query all the kinds of events\(^4\) that are possible under a copy of the real control-state, or under some hypothetical one, analogously to the possible-event query. The difference is that here it is the user who can specify the control-state to be used. A query request is conveyed when the user presses the *query* button right below the *repertoire* label. The list to the right of label *repertoire* in Figure 9 shows the result of such a query, and choosing an item of the list has it pasted to the *event* text-field for ease of editing in using Feature (b).

**Feature (b):** As an regulated event to be test-evaluated by the controller, the user can enter any type (including certified, obligationDue, and exception), in the *event* text-field at the top of the sub-window. Note that the term entered must follow the complete syntax of a regulated event; e.g., if it is a sent event, it must start with functor sent, and must contain the sender, the message body, and the intended recipient. Pressing the *test* button sends a message, with the regulated event specified in the *event* field as its payload, to be test-evaluated by the controller, under the control-state as explained below.

The result of the test-evaluation is to be reflected as follows: First, the *result* text-field right below *test* button shows either success or failure, depending on the outcome of the evaluation. If the evaluation is successful, a few other updates are made: (1) the *rule* text-area, to the right of

\(^4\)However, events of obligationDue and those of exception are excluded since these events occur depending on the past execution of certain primitive operations, which is currently out of the scope of the testing.
the test button, gets the text of the rule that has yielded the non-empty ruling (with the associated comment, if any, in the law); (2) the do-list text-area right below the result field receives the list of do goals of primitive operations that represents the non-empty ruling; and finally (3) the CS text-area gets the new control-state that reflects the non-empty ruling of this test-evaluation, based on the control-state specified in the query.

Selecting between the real control-state and a hypothetical one: Just above the text-area labeled CS, to the right of this label, there are two radio buttons labeled: real CS and as below. The selection over these two makes a difference in the control-state that the testing mode of the controller uses to derive all possible regulated events in Feature (a), or to test-evaluate the specified event in Feature (b). If the real CS button is chosen, such computation is based on a copy of the real control-state. However, if the as below button is chosen, the content of the text-area right below is used as the control-state. Thus, the user can edit the content any way he likes, to set up a hypothetical control-state, where the text in this area must be in the form of a Prolog-list (though new-line characters are allowed), enclosed with a pair of brackets ([I and J]). The selection between these two radio buttons is to be set on as below whenever a successful test-evaluation of Feature (b) is reflected in the testing window, which is meant to be useful for the user to conduct a series of testing queries. (Whenever the user wants to return to the real control-state to base the testing on, he can just select back the real CS button.)

Because the interface provides an option to carry out testing based on (a copy of) the current (real) control-state, as seen above, the user could learn some aspects of the control-state through such testing. Therefore, the availability of testing is (again) regulated by the law, in a manner we explain next.

Control by the law over the testing: There are two LGI primitive operations used to regulate the availability of testing: enterTest and exitTest. Operation enterTest lets the agent start using the testing mode of the controller, with which the actor can test the ruling of various regulated events under possibly hypothetical control-states. The testing mode of the controller is provided by maintaining hypothetical control-state, completely disjoint from the real one used in the communal regulation of this agent. In addition, upon the execution of enterTest, the controller sends the following message, as a cue, to the actor:

\[\text{[arrived(testModeEntered),from(controller)]}\]

Operation exitTest is the companion of enterTest, whose execution makes the testing mode of the controller unavailable to the actor, while sending the following cue to the actor:

\[\text{[arrived(testModeExited),from(controller)]}\]

Some ways to use these primitive operations in a law are seen in Figure 1. In particular, operation enterTest is executed by rule \(R1\), only when the actor sends an atom humanUser as part of the arguments to the adopted event. Operation exitTest is executed by rule \(R4\), when the actor sends an exitTest message. Note that the regulation over the availability of the testing implemented by these rules is possibly the least restrictive, in that any member who desires to use it is given the permission as soon as he enters the community, and may use it as long as he wants. The tailoring of this kind of regulation is left to individual law writers.

Finally, note that if and only if the user of the interface is permitted to use the testing mode, i.e., primitive operation enterTest has been executed by the controller (while exitTest has been absent), the query button in Feature (a) and the the test button in Feature (b) are selectable.
6 Summary

In an open distributed system, the circumstances in which components operate may change dynamically. Thus, in order to facilitate effective operation of these components, it is important to supply them with timely information to characterize such circumstances.

In this chapter, we focused on the following two kinds of queries that produce such information: (a) “What can this member do?” (which we call the possible-event query); and (b) “What happens if a certain event occurs to this member?” (the testing query). We implemented a mechanism to process these queries in the context of LGI, and demonstrated it by presenting a user interface that utilizes the information derived from such queries, to support human members of a community under a certain law. The invocation of these queries is regulated by the law of the community, which allows for application specific control over which members can get the information based on the queries, under what conditions.

By the result of possible-event queries, the interface updates the complete list of messages that the user can send at the moment, called the repertoire. Through series of testing queries, the user can examine hypothetical state transitions under the law. Thus, in addition to better informing a user for making decisions on his next actions, the user interface can be used to “debug” the law, and to help “novices” in the community learn the law.

A The Output of LGI Primitive Operation show

To be concise, we use an example to explain the format of the output message from the show operation. Assuming that in the law, the corresponding show operation has two types of regulated events certified and sent in its possibleEvent parameter, a message to carry the disclosed information to the actor is of the following format (pev for “possible event”, and new-line characters are inserted for the sake of presentation):

\[
\text{[[pev(certified(X,certificate(issuer(I),subject(S),attributes(L))),'rule'), ...],}
\text{[pev(sent(S,M,D),'rule'), ...],}
\text{CS('comma-separated-terms')]}\]

I.e., it is a list of certain terms, and each component term is either a list of pev terms, or a term with functor CS (in capital); the former corresponds to the possibleEvent parameter of the operation, and the latter to the cs parameter. In the case of such a term being a list of pev terms, each is made of the same type of regulated event and the rule of the law that yields non-empty ruling on that event, along with a comment in the law, if any. (The rule and the associated comment are enclosed in a pair of double-quote ("**) symbols.) In the case of the CS term, the terms being disclosed from the control-state are placed as arguments of this term, separated by a comma. (Thus, if there are \(n\) terms being disclosed, the arity of the CS term is \(n\) as well.)

B The API for the Testing Mode of the LGI Controller

We explain a method test of class moses.member.Member, provided for a program to interact with the testing mode of the controller. As mentioned in Section 5, this mode of the controller becomes available only when primitive operation enterTest is executed, and becomes unavailable when primitive operation exitTest is executed. Unless the mode is available, the invocation of the following method causes no effect except consuming computing resources (see Section 5 for the pair of cues given by the controller to tell the availability).
• void test(String event, String cs): sends the controller a testing query to (test-)evaluate regulated event event under control-state cs. The cs should be bound to either a Prolog list or a keyword use_current; in the latter case, the real control-state for the regular ruling (as opposed to the test ruling) available at the moment is copied, and used for the query processing.

Reply from the controller: After test-evaluating the event based on the cs, the controller replies asynchronously with a message in the following form:

\[
\text{test_reply('event','success-failure','new-cs','do-list','rule')}
\]

where 'event' is the first argument supplied to the above method invocation; 'success-failure', either success or failure, indicates the result of the evaluation requested. If the evaluation is successful, the rest of the reply message contains the following:

- 'new-cs': a Prolog-list that represents the control-state, with the non-empty ruling of this test evaluation reflected to the cs argument given in the above method invocation;
- 'do-list': the non-empty ruling produced by the testing mode, less the changes to the control-state (which is reflected in 'new-cs' above);
- 'rule': the rule of the law that has yielded the non-empty ruling, with comments, if any in the law. The entire string is “double-quoted” (i.e., the first and the last characters are both ").

If the evaluation result is failure, 'new-cs' and 'do-list' are both \[
\] (i.e., the empty list), while 'rule' is NA (for “not applicable”).

To invoke a query to get all possible events: For the first argument of the above method invocation, one can specify a keyword, query, instead of a regulated event. Given such a request, the controller’s testing mode generates a list of all kinds of regulated events that would lead to a non-empty ruling, based on the control-state specified in the second argument of the method invocation. (This corresponds to Feature (a) of the user interface in Section 5, while having the event argument be a well-formed regulated event, as described above, corresponds to using Feature (b).) Having produced such a list, the testing mode replies (asynchronously) with a message of the following form:

\[
\text{testQuery_reply('answer_list')}
\]

where 'answer_list' carries the generated list of events in the following form (pev for “possible event”, and new-line characters are inserted for the sake of presentation):

```
[[pev(certified(X,certificate(issuer(I),subject(S),attributes(L))),'rule'), ...],
 [pev(sent(S,M,D),'rule'), ...],
 ....]
```

I.e., it is a list of lists, and each component list consists of pev terms for the same type of regulated event. Each pev term has the same form as the one described in Appendix A. Note that two event types, obligationDue and exception, are not the target of such query.