Rewriting Logic Approach to Modeling and Analysis of Client Behavior in Open Systems

Shin Nakajima, Masaki Ishiguro, Kazuyuki Tanaka

Rewriting Logic Approach to Modeling and Analysis of Client Behavior in Open Systems

Shin Nakajima[1], Masaki Ishiguro[2], and Kazuyuki Tanaka[3]

[1] National Institute of Informatics, Tokyo, Japan
[2] Mitsubishi Research Institute, Inc., Tokyo, Japan
[3] Hitachi Software Engineering Co., Ltd., Tokyo, Japan

Abstract. Requirements of open systems involve constraints on clients behavior as well as system functionalities. Clients are supposed to follow policy rules derived from such constraints. Otherwise, the system as a whole might fall into undesired situations. This paper proposes a framework for system description in which client behavior and policy rules are explicitly separated. The description is encoded in Maude so that advanced analysis techniques such as LTL model-checking are applied to reason about the system properties.
Keywords: Requirements, Validation, Model-Checking, Maude

1 Introduction

Embedded and ubiquitous computing systems are typically open, and their functional behavior has much impact from the environment [5]. It involves human behavior if the systems are user-centric. Unfortunately, thoroughly anticipating client behavior is a hard task and a certain assumption on the client behavior is usually made. Clients are expected to follow an operation guideline to consist of policy rules, which is a concrete form of such assumptions.

In general, a relationship between requirements and system specification is concisely summarized in the *adequacy* property [4]; $D \wedge S \Rightarrow R$, where R and S represent requirements and system specification respectively. D is *Domain Knowledge* that provides background information and is an assumption on the system. D, in open systems involving clients, can be elaborated into $D_c \wedge D_p \wedge D_o$. D_c and D_p refers to the domain knowledge on client behavior and policy rules respectively while D_o represents other domain knolwedge and is omitted here for brevity.

Requirements R may usually consist of typical usecases to assume that clients behave in a *regular* manner ($D_c \wedge S \Rightarrow R$). Clients, however, sometimes show irregular behavior. Some may be careless, and a few might even try to cheat the system or obstruct other clients. Such behavior could result in undesired consequence. To avoid such situations, the policy rules (D_p) are introduced to constrain the client behavior. There is, however, another problem; it cannot be ensured whether clients follow the policy rules or not. In a word, the assumption made at the time of the system development is no longer valid at the time of

operation. It is desirable to provide means fo studying possible outcomes at the development time.

In this paper, we will discuss an approach to requirements modeling of open systems involving human clients. The proposed method allows us to represent and reason about the system specification at early stages of the development. It makes it possible such an analysis as whether a certain irregular client behavior might lead to situations against the requirements. In particular, the contribution of this paper is summarized as follows.

1. it proposes a framework for system description in which clients behavior (D_c) and policy rules (D_p) are explicitly separated,
2. the description is encoded in Maude [3], thus allowing automated analysis.

The rest of the paper is structured as follows. Section 2 introduces Guiding System as a motivating and running example. Section 3 presents a formalization approach to the problem and a role of model-checking in the analysis. Section 4 explains the encodings in Maude. Section 5 shows a case study, which is followed by discussions and conclusion in Section 6.

2 Guiding System : A Motivating Example

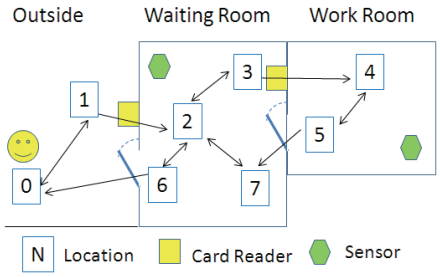

Fig. 1. Guiding System

A Guiding System is used so as to illustrate the problem and approach. In Figure 1, the Guiding System is a user-navigation system for clients to access particular resources. Clients arriving at the system firstly check with their ID Card if they have been registered. They then wait in a Waiting Room until receiving a call to allow them to exclusively use the resource. The system keeps track of the status information of the resource usage and waiting queue of clients. When a resource is free, the system will call a client in the Waiting Room. The client moves to the Work Room where he can access the resource.

Except the above regular scenario, a lot of cases could happen. A client might try to go to the Work Room even if he was not the one to be called. A client

might walk around in the Waiting Room disturbing others by blocking the door to the Work Room. Some might be absent when he is called, and would not use the resource at all.

The Guiding System might detect some of the anomalous situations if certain fancy gadgets are available. For example, all the client locations can be monitored if every person has a remote badge that can be sensed with wireless communication network. Precise location of each client is surely monitored. It, however, will increase the cost of the system.

Alternatively, the Guiding System is used with a certain running policy, which poses some constraints on the client behavior. In the real world, an officer works near the system and he periodically checks to make sure all the clients behave in a regular manner. For example, he will tell a client who walks around in the Waiting Room, "please sit down and wait for a call." The system can be defined without being over-engineered since the clients are assumed to follow mostly the regular scenario. The implicit assumptions on the client behavior are made explicit in the form of policy rules.

3 Formal Analysis of Client Behavior

3.1 Basic Computation Model

As discussed in Section 2, what will be specified consists of many entities executing concurrently. Some entities constitute the Guiding System, and clients are autonomous to move in the rooms. Furthermore, a hypothetical component to enforce the policy rules executes independently from other entities. Such observations lead us to adapting a modeling method of using a set of communicating state-transition machines. Additionally, we have introduced a notion of Location Graph to reason about the client movement. Below are some definitions to constitute the basic model.

[**Extended Mealy Machine**] Extended Mealy Machine (EMM) M is intuitively an object to have its own internal attributes and its behavioral aspect specified by a finite-state transition system. They communicate with each other by sending and receiving events. An EMM M is defined as a 6-tuple

$$M = (Q, \Sigma, \ \rho, \delta, q_0, \mathcal{F})$$

Q is a finite set of states, Σ is a finite set of events, q_0 is an initial state, and $\mathcal{F}$ is a finite set of final states. Variable map ρ takes a form of $Q \rightarrow 2^V$ (a set of attribute values). Transition relation δ takes a form of $Q \times A \times Q$. The action A has further structure to have the following functions defined on it; $in : A \rightarrow \Sigma$, $guard : A \rightarrow L$, $out : A \rightarrow 2^{\Sigma}$ where L is a set of predicates.

Operationally, a transition fires at a source state q to cause a state change to a destination p if and only if there is an incoming event $in(A_q)$ and $guard(A_q)$ is true. In accordance with the transition, possibly empty set of events $out(A_q)$ are generated. Furthermore, *update* function is defined on A to change ρ, which modifies the attribute values stored in M.

[**Configuration**] Configuration constitutes a set of EMMs and a communication buffer to store events (Σ) used for asynchronous communication between EMMs.

[**Run**] Run of a EMM$^{(j)}$ ($\sigma^{(j)}$) is a sequence of states ($Q^{(j)}$) which are obtained by a successive transitions from the initial state ($q_0^{(j)}$) following the transition relation $\delta^{(j)}$. A run of Configuration is a possible interleaving of $\sigma^{(j)}$ for all EMMs, taking into account of the event communications among them.

3.2 Location and Client Behavior

[**Location Graph**] Location Graph is a directed graph (N, E). N is a set of locations, and a location is where clients can occupy; clients walk from a location to another. Since clients walk along the edges between locations, nodes N are connected with edges E, which together form a directed graph.

As seen in Figure 1, clients use doors to enter and leave a room. Such doors are considered to attach with particular edges where clients can move along. Location Graph, in this sense, represents the structure of the rooms that the system works on.

Note that location in the graph does not correspond to a physical place in the real world, but represents an abstract place corresponding to a set of them. It is abstract in that client behavior from any of the pysical places in the set is not distinguished.

[**User Machine**] User Machine (UM) is a special EMM, which has a default attribute to represent its location (N of Location Graph). Furthermore, client behavior is designed to follow a simple template. First, it is in *Mediate* state to determine which location it moves to next (*a trial step*). Second, a state transition is fired to *Walk*, in which the client actually changes its location. Third, another transition puts the client to *Action* state to take various actions such as swiping his ID Card or opening doors.

In each state, UM takes a sequence of *micro-transitions* between *micro-states* to show application-specific behavior. Therefore, a run for User Machine consists of micro-states, but a sequence of Locations is what we want to know.

[**Location Projection**] Location Projection ξ is a function to extract a pair of client Id and its Location from state Q of User Machine EMM; $\xi : Q \rightarrow \text{ClientId} \times N$. Note that ξ can work only on User Machine. It returns ϵ (*null*) for all the other EMMs.

[**Location Trail**] Location Trail is a sequence of location projection, which is obtained from a (configuration) run to be a sequence of $\xi(\sigma^{(j)})$.

3.3 Policy Rules

We focus on reasoning about client movements, namely how each client moves around. Policy rule is defined on Location Trail.

[**Policy Rule**] Policy Rule monitors clients movement and enforces them to move as anticipated. It keeps track of Location Trail, and it checks whether a client trial step is admissible or not. A Policy Rule takes the following form.

if $filter(x,\ l)$ **then** $action(x)$

where x and l refer to a client and a location trail respectively. Currently, three forms of actions are considered; (a) **a permission rule** to allow the client as he wants to do, (b) **a stop rule** to instruct the client not to take any action, and (c) **a coercion rule** to make the client to move as the policy rule instructs.

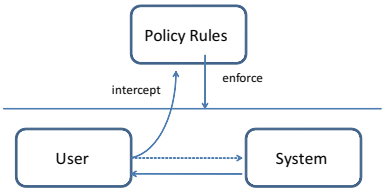

Fig. 2. Two-tiered Framework

[**Policy Enforcer**] Policy Enforcer is an executing entity who enforces a given set of Policy Rules. Policy Enforcer is represented as a special EMM, but executes at a kind of *meta level*. Since it uses Location Trail and checks every trial step, a new architectural mechanism is needed to monitor all the clients movement trial. In Two-tiered Framework shown in Figure 2, user trial step is conceptually intercepted and checked againt Policy Rule, and then its result is enforced. The client moves as the policy rule specifies.

3.4 Analysis Problem

Fig. 3. Lasso Shape

Policy Enforcer, introduced in the previous section, is defined as an entity to monitor the client behavior at runtime to see whether some undesired situation might happen. It basically checks if Location Trail for a particular client falls into anomalous loop, which shows that the client moves indefinitely between some particular locations and that he never goes to a desired destination. When a client rests at a particular location never to move, the anomalous situation is detected as a loop to contain a single location only.

Such anomalous situation is detected as an existence of Lasso Shaped Location Trail (Figure 3). The analysis problem of detecting such a loop can be

considered as a model-checking problem [2]. Let $at(C, L)$ be a predicate to represent that a client C is located at a location L. A model-checking problem with given client $u1$ and location $L4$ is written below.

$$Configuration \models \texttt{[]<>}(at(u1, L4))$$

The property to check is read "it is always the case that eventually $at(u1, L4)$ becomes $true$." If the client $u1$ behaves in a irregular manner to form a loop not to reach the location $L4$, a counterexample trace takes a form of the lasso shape as expected.

In other word, the adequacy property [4], $D_c \wedge D_p \wedge S \Rightarrow R$, is here considered as a model-checking problem of $D_c \ \wedge S \models r$ where r is an LTL (Linear Temporal Logic) formula chosen from R. If r is satisfied, then D_p is not needed. Otherwise, a counterexample trace is obtained, from which policy rules (D_p) are derived. Namely, with D_p so constructed, the relationship $D_c \wedge D_p \wedge S \models r$ holds.

4 Rewriting Logic Approach

4.1 Rewriting Logic and Maude

Rewriting logic is proposed by Jose Meseguer, which follows the tradition of algebraic specification languages [3]. The logic extends order-sorted algebra of OBJ3 to provide means to describe state changes. Maude, an algebraic specification language based on Rewriting logic, is powerful enough to describe concurrent, non-deterministic systems. Such systems can be symbolically represented with appropriate level of abstraction. Furthermore, Maude provides advanced means to analyzing properties of system with various state-space search methods such as bounded reachability and LTL model-checking.

4.2 Encoding Extended Mealy Machine in Maude

With Maude, the artifacts that we are interested in are modeled as a collection of entities executing concurrently. Their functional behavior is expressed in rewriting rule, which takes a form of "$lhs \rightarrow rhs$ **if** $cond$." Transition relation δ from q to p in EMM (Section 3.1) is described by rewriting rule;

$$in(A_q) \ \langle \ \rho_q \ \rangle \quad \longrightarrow \quad \langle \ \rho_p \ \rangle \ out(A_q) \quad \textbf{if} \quad guard(A_q)$$

where $\rho_p = update(\rho_q)$.

Encoding EMM in Maude requires further modules to define. Following example is used to show how an EMM is described. Firstly, MACHINE module is a functional module (`fmod`) that provides a basic syntax (term) of EMM.

[4] D_o is ignored for simplicity.

```
fmod MACHINE is
 sort Machine . sort StateId .  sort TypeId .
 protecting QID . protecting MID .  protecting ATTRIBUTES .
 subsort Qid < StateId .  subsort Qid < TypeId .
 op <_:_|_;_> : MachineId TypeId StateId Attributes -> Machine [ ctor ] .
endfm
```

The module MACHINE defines the sort Machine and imports the sorts of Qid (a built-in primitive sort for representing unique Id) and Attributes. It has a constructor which includes unique Id for the instance, the type of this instance, the state and the attributes. Here is a simple example of Machine, a DOOR.

```
  extending SOUP .
  op makeDoor : MachineId -> Machine [ctor] .
  var D : MachineId .
  eq makeDoor(D) = < D : 'Door1 | 'lock ; null > .
```

The constructor `makeDoor` is meant to have an initialized instance of Door Machine. Their dynamic behavior, namely state changes, is described in terms of a set of rewriting rules. The state-transition diagram in Figure 4 shows its simple behavior, which is encoded by the rewriting rules below. Each rewriting rule corresponds to a transition in the diagram.

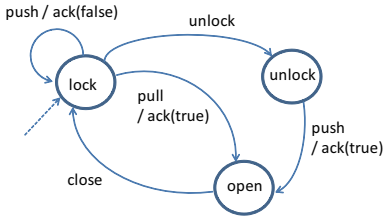

Fig. 4. State Transition Diagram of Door

```
vars D U : MachineId . var R : Attributes .
 rl [1] : push(D, U) < D : 'Door | 'lock ; R >
    => < D : 'Door | 'lock ; R >  ack(U, false) .
 rl [2] : unlock(D) < D : 'Door | 'lock ; R >
    => < D : 'Door | 'unlock ; R > .
 rl [3] : push(D, U) < D : 'Door | 'unlock ; R >
    => < D : 'Door | 'open ; R > ack(U, true) .
 rl [4] : pull(D, U) < D : 'Door | 'lock ; R >
    => < D : 'Door | 'open ; R > ack(U, true) .
 rl [5] : close(D) < D : 'Door | 'open ; R >
    => < D : 'Door | 'lock ; R > .
```

A client generates a `push` event to open a door. If the door is in `lock` state, the rule `[1]` is fired. The door remains shut and sends an `ack(fail)` event to the client. Rule `[2]` shows that the door can be unlocked by the card reader. Alternatively in rule `[3]`, when the door is in `unlock` state, its state is changed from `unlock` to `open` as the door receives a `push` event. The other rules can be understood in a similar manner.

5 Case Study

5.1 Modeling

Location Graph Figure 5 is an instance of location graph to represent an abstract view of the example in Figure 1. Each location has a capacity attribute to show how many clients are allowed at a time. For example, L1 is where a client swipes his ID Card and its capacity is 1. On the other hand, more than one clients are waiting at L2, thus the capacity of L2 is *many*.

Clients start from L0 and move along the edges depicted with the arrows. At each location, clients may take some actions and they proceed to the next location if prescribed conditions are satisfied.

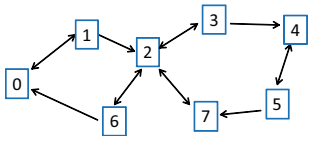

Fig. 5. Location Graph Example

Guiding System The Guiding System consists of various entities constituting the room (Figure 1), each being defined as an EMM. Figure 6 illustrates all the components and interaction between them. Guide System or GSystem is a main component to provide user-navigation functions for clients to access the resources in the Work Room. It accepts *abstract* events from FrontEnd and makes announcement to let a client enter the Waiting Room.

Door1 is a door to enter the Waiting Room, while Door2 is the one to enter the Work Room. Both is usually locked so that clients must unlock it. The door, however, can be open from the inside to leave even when the door is locked.

CardReader1, when a client swipes his ID Card, checks a backend database to see whether he is a valid registered client. When he is found registered, CardReader1 generates a $unlock(Door1)$ event to unlock the door. CardReader2 also checks the client ID Card when he is called to enter the Work Room. CardReader2 does not access the Database, but uses the valid client information *set* by GSystem. In this way, only the called client is certified.

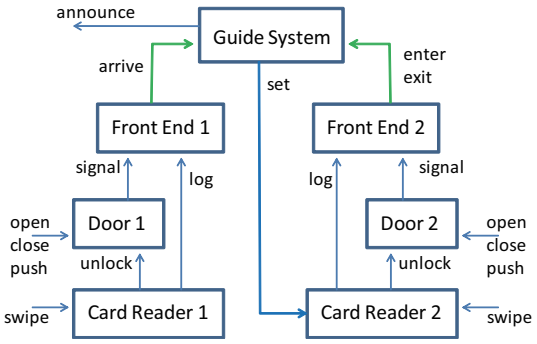

Fig. 6. Guiding System Components and Their Interactions

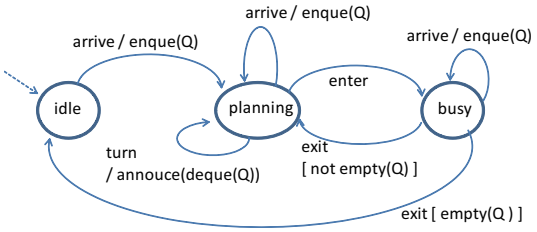

Fig. 7. State Transition Diagram of GSystem

Figure 7 illustrates the state transition diagram to show behavioral specification of GSystem. Its behavior is defined as a set of responses to three abstract events; *arrive*, *enter*, and *exit*. Their informal meanings are

- $arrive(U)$: a client U enters the Waiting Room from the outside,
- $enter(U)$: a client U enters the Work Room from the Waiting Room,
- $exit(U)$: a client U leaves the Work Room.

Initially, GSystem is in `idle` state and changes its state to `planning` when an $arrive(U)$ event is generated to indicate that a paricular client U arrives. In the course of the transition, U is enqueued and thus is later consulted for GSystem to determine the next client to enter the Work Room. In view of client, he is put in the queue when he arrives and waits for a call by GSystem.

After generating $announce(U)$ event, GSystem changes its state to `busy` as it receives an $enter(U)$ event. The event indicates that the called client enters the Work Room. All the $arrive(U)$ events are queued while GSystem is in either `planning` or `busy` state. Upon receiving an $exit(U)$ event, GSystem knows that the Work Room is no longer occupied, and it moves to `planning` state again if the queue is not empty. Alternatively, it jumps into `idle` state if the queue is empty.

Although the behavior of GSystem can be defined in terms of three abstract events mentioned above, the instruments equipped with the room do not generate those. Instead, they generate primitive events. For example, CardReader generates $log(U)$ events when U is recognized as a registered client. Door generates *open* and *close* events. There should be a certain mechanism to bridge the gap between the primitive and abstract events.

FrontEnd shown in Figure 6 is responsible for the translation. In particular, FrondEnd1 takes care of the instruments at the entrance of the Waiting Room. It receives $log(U)$, *open*, and *close* events in a specified sequence, and generates $arrive(U)$ event to GSystem. FrontEnd2, looking at events generated at the border of the Waiting Room and the Work Room. It generates $enter(U)$ event from $log(U)$, *open*, and *close*. Furthermore, it generates $exit(U)$ event from *push* (for leaving the Work Room) and *close*.

5.2 Analysis

Tha Maude description developed so far is composed of 32 modules in about 1,000 lines of codes. About 60 % of codes are responsible for dynamic behavior, namely rewriting rules. Below present some results of analysis conducted with MOMENT Maude 2.4 [1] running on Panasonic CF-W7 under Windows/XP.

Validation in Maude Firstly, an initial state of the Guiding System shown in Figure 1 is constructed. Secondly, some appropriate User Machine(s) are added to form an initial state. Here, we define a User Machine so that it can initiate its execution by receiving an event *start(U)*. Furthermore, a client follows a regular behavior to leave L0 and return to the same place after doing his job in the Work Room. Below, execution results are represented as Location Trail, traces of a tuple consisting of User and Location, (U, L).

The first example shows a trace of two clients, u1 and u2. The initial state includes two initial events *start(u1)* and *start(u2)*. Here is one possible trace taken from a result of rewriting run (`reduction`) in Maude.

```
('u1, 'L1), ('u1, 'L2), ('u2, 'L1), ('u1, 'L3), ('u2, 'L2), ('u1, 'L4),
('u1, 'L5), ('u1, 'L7), ('u1, 'L6), ('u2, 'L3), ('u1, 'L0), ('u2, 'L4),
('u2, 'L5), ('u2, 'L7), ('u2, 'L6), ('u2, 'L0)
```

As the trace shows, u2 moves to L1 after u1 goes to L2 since L1 is a place to use Card Reader1 and only one user can occupy the location at a time. It also shows that u2 stays at L2 until u1 moves to L6.

The second example consists of a regular client u2 and another client v1. V1, initially at L2, disturbs other users by moving between L2 and L3 repeatedly. Since L3 is a place to allow only one user, no other user can move to L3 while v1 occupies the place. Here is a trace.

```
('u2, 'L1), ('v1, 'L3), ('v1, 'L2), ('v1, 'L3), ('u2, 'L2), ('v1, 'L2),
('v1, 'L3), ('v1, 'L2), ('u2, 'L3), ('u2, 'L4), ('v1, 'L3), ('v1, 'L2),
('u2, 'L5), ('v1, 'L3), ('u2, 'L7), ('v1, 'L2), ('v1, 'L3), ('u2, 'L6),
('v1, 'L2), ('u2, 'L0)
```

Regardless of v1 behavior, the client u2 can move to L2 because more than one clients can share the location. U2, however, has to wait at L2 before going to L3 even he is called. The client v1 moves between L2 and L3, and L3 becomes available occasionally.

LTL Model-Checking In Section 3.4, we discussed that the monitoring client behavior in view of the Policy Enforcer was a model-checking problem. Model-checking, of course, can be used as *debugging* system behavior as well. Actually, we could find one fault in the original description of the Guiding System where two clients were involved in the scenario. The fault was not revealed just by the validation with `reduction`.

A conflict is occurred at the Door1 (Figure 1). A client u1 at L6 tries to get out of the Waiting Room to push the Door1. Another client u2 swipes his ID card at L1, which is followed by an `unlock` event sent to the same door. It results in a conflict of two events, `push` and `unlock`.

In order to avoid such conflicts, we introduced another Door3, which was used solely as an exit from the Waiting Room. Door1 is now for the entry only. Althought installing a new Door might be a solution to increase the cost of the system, we use the description with Door3 in the following analysis for simplicity. It is especially important to study such design alternatives at this abstract level with the formal analysis of Maude description.

We now demonstrate how model-checking is conducted for the problem relating to Policy Enforcer. Imagine that a client u1 shows irregular behavior. A regular scenario assumes that a client, when he is called at L2, walks to L3 and swipes the ID Card to unlock the door. The client u1, however, walks to L3, but returns to L2 without taking any action at L3, which is repeated indefinitely. Let $goAound(u1)$ be a proposition to denote that the client u1 returns to L0, the outside of the system. The property `[]<>`$(goAound(u1))$ is not satisfied and an obtained evidence is a counterexample, `[L0, L1, L2, L3, L2, L3, ...]`. Such an infinite sequence is abbreviated as `[L0, L1, (L2, L3)*]`, where (...)* represents a loop in the lasso. Policy Enforcer may apply the stop rule (Section 3.3) to the client u1.

Next, consider a case where a non-determinisitc behavior at L3 is added to the above client u1. When he is at L3, he chooses non-deterministically either to walk back to L2 or to enter the Work Room by following prescribed scenario. Model-checking of the property `[]<>`$(goAound(u1))$ returns the same lasso-shaped counterexample as above. We can, however, ensure that there is at least one witness trace for the client to enter the Work Room. The property $\neg$`<>`$(at(u1, L4))$ is not satisfied (namely `<>`$(at(u1, L4))$ is satisfied), and the obtained evidence is a witness of `[L0, L1, L2, L3, L4]`. From the model-checking result, we can derive such a policy rule (D_p) as "clients, once called in the waiting room, should eventually enter the work room." It is an instance of the coersion rule (Section 3.3).

These two example scenarios can be explained in the following ways. Firstly, for a client who shows an irregular behavior (D_c^i), we have $D_c^i \wedge S \not\models$ `<>`$(at(u1, L4))$.

Secondly, if the client adds non-determistic behavior (D_c^n) with an appropriate policy rule (D_p) enforced, we have $D_c^n \wedge D_p \wedge S \models \Diamond(at(u1, L4))$.

In summary, from the results of $D_c \wedge S \not\models r$, we construct D_p such that $D_c \wedge D_p \wedge S \models r$. Because the aim of D_p breaks the loop in the lasso, a possible method is to adapt the notion of ranking functions [6]. In the most simplifed form, a ranking function rk is monotonically decreasing with a lower bound lb. D_p is defined to break the lasso loop when rk reaches lb. Policy Enforcer is defined so that it adapts a counter as rk and applies a coersion rule to break the loop when the counter reaches the specified value. Currently, D_p is obtained by a manual inspection of the generated trace.

6 Conclusion

We have proposed a rewriting logic approach to modeling and analysis of client behavior. Firstly, we have discussed the importance of separating human client behavior from policy rules. With a conceptual two-tiered framework, the client can be checked whether his behavior faithfylly follows the policy. Second, such analysis can be considered as a model-checking problem, and hense studying alternative design in the presence of clients is possible at the time of system development. Policy Enforcer in our proposal is similar to the one discussed by Schneider [7], in which he proposes security automata, a variant of Buchi automata. Since it is encoded in EMM, Policy Enforcer is a finte-state automaton. Relation to model-checking problem has not been discussed so far.

In this paper, a simple Guiding System was used to illustrate the idea. The proposed method can be applied to any application system where policy rules are explicitly mentioned to enforce client behavior. Applying the method to such application systems is one possible area to pursue. Last, as a theoretical aspect of the method, future work includes automated discovery of ranking function rk for D_p from the analysis of counterexamples of $D_c \wedge S \not\models r$.

References

1. MOMENT Web Page. `http://moment.dsic.upv.es/`
2. E. M. Clarke et al. *Model Checking*. MIT Press 1999.
3. M. Clavel et al. *All About Maude – A High-Performance Logical Framework*. Springer 2007.
4. M. Jackson. *Requirements & Specifications*. Addison-Wesley 1995.
5. M. Jackson. The Role of Formalism in Method. In *Proc. FM'99* pages 56, 1999.
6. Y. Kesten and A. Pnueli. Verification by Augmented Finitary Abstraction. *Information and Computation*, Vol.163, No.1, pages 203-243, 2000.
7. F.B. Schneider. Enforceable Security Policies. *Transactions on Information and System Security*, Vol. 3, No.1, pages 30-50, 2000.