社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

ポリシー規則と振る舞い仕様の分離への書き換え論理アプローチ

シャオシ　ドン[†]　中島　震[†]

[†] 国立情報学研究所　〒 101–8430 東京都千代田区一ツ橋 2–1–2
E-mail: †nkjm@nii.ac.jp

あらまし　オープンなシステムの開発では、利用者の振る舞いに仮定を置くことで、システム要件を簡単化できることがある。利用者は予め決められた運用規則（ポリシー規則と呼ぶ）に従うことが期待される。これに反する振る舞いによってシステム全体が不具合に陥るかもしれない。また、システム運用開始後にポリシー規則が変更される可能性もあり、その場合に対してもシステム要件は堅固であるか開発上流工程で知っておきたい。本稿では、システム機能、利用者の振る舞いに加えて、ポリシー規則を明示的に分離する方法を提案する。また、書き換え論理の方法に基づいて分離の一般的なフレームワークを論じる。
キーワード　システム要件、ポリシー規則、妥当性確認、ラピッドプロトタイピング

Rewriting Logic Approach to Separating Policy Rules from Behavioral Specification

Xiaoxi DONG[†] and Shin NAKAJIMA[†]

[†] National Institute of Informatics　2–1–2 Hitotsubashi, Chiyoda-ku, Tokyo, 101–8430 Japan
E-mail: †nkjm@nii.ac.jp

Abstract　Requirements of open systems involve constraints not only on system functionalities but also on client behavior. Ideally, clients are supposed to follow the policy rules derived from such constraints. However, when this assumption is not true, the system might fall into abnormal status. Furthermore, policy rules may be changed even after service-in, and knowing before-hand whether the system is robust or not in accordance with such changes is desirable. This paper proposes a framework for system description in which client behavior and policy rules are explicitly separated, which is encoded in rewriting logic proposed by J. Meseguer.
Key words　System Requirements, Policy Rules, Validation, Rapid Prototyping

1. Introduction

Defining precisely the environment in which computing systems are used is important at early stages of system development [4]. It is especially true for systems to involve users. Unfortunately thoroughly anticipating user behavior is a hard task. Some users may be careless, and a few might even try to cheat the system or obstruct other users. These behavior could result in undesired consequences. The system can be equipped with fancy gadgets to detect the anomalous situations on-the-fly. This approach, however, may result in an over-engineered system with exceeding budget. Alternatively, the system assumes the users to follow a certain set of policy rules without showing *bad* behavior. With such assumptions, the system design can be less complicated.

Currently, the system requirements are studied together with proper policy rules constraining users behavior. It, however, may happen that such policy rules are changed after the system has been completed or even after coming into operation. Policy rules are constraints on the user behavior. Their changes do not seem to incur further development cost, which is not true. The system had been developed under certain assumptions which were assembled into a set of policy rules. Therefore, a slight change in policy rules may have much impact on the system behavior. It is desirable to check before-hand whether the system is robust to such changes. In a word, representing the policy rules explicitly is mandatory for a long-term system evolution.

This paper discusses how the policy rules play an essential role in open systems that involves users. It proposes a

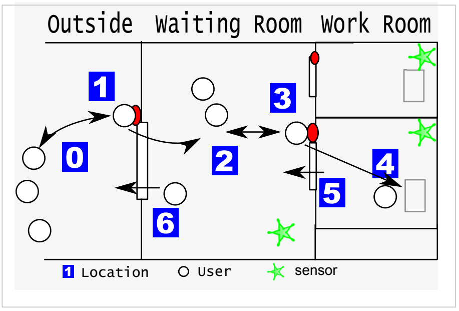

図 1 Guiding System Example

framework for system description in which user behavior and policy rules are explicitly separated. Furthermore, an example case following the proposed idea is encoded in Meseguer's rewriting logic [2] so that various analysis, such as validation and property checking, is possible.

2. Guiding System : A Motivating Example

In this paper, we use a Guiding System so as to illustrate our idea (Figure 1). The Guiding System is a user-navigation system for clients to access particular resources. Clients arriving at the system check firstly if they have been registered. They then wait in a waiting room until receiving a notice to allow them to exclusively use the resource. The system keeps track of the status information of resource usage and the waiting queue of clients. When a resource is free, the system will call a particular client in the waiting room. The client moves to the work room where he can access the resource.

Except the above regular scenarios, 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 even if he is called, and he 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 clients locations can be monitored if they have a remote badge that can be sensed by using wireless communications. It, however, will result in an over-engineered system with exceeding budget.

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 say to a client who walks around in the waiting room, "please sit down and wait for the announcement." The system can be defined without being over-engineered since the clients are assumed to follow mostly the regular scenarios. The implicit assumptions on the client behavior are made explicit in the form of policy rules.

3. Two-tiered Framework

The system, as discussed in Section 2., 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 observation leads 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 clients movement. Below are some definitions to constitute the model.

[Extended Mealy Machine] Extended Mealy Machine (EMM) M is intuitively an object to have its own internal attributes and to have its behavioral aspects specified by a finite-state transition system. They communicate with each other by sending and receiving events. 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. A variable map ρ takes a form of $Q \rightarrow 2^V$ (a set of variables). A transition relation δ takes a form of $Q \times A \times Q$. The action A has further structure to have the following functions defined; $in : A \rightarrow \Sigma$, $guard : A \rightarrow L$, $out : A \rightarrow 2^{\Sigma}$ 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, which changes 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$_j$ (σ_j) is a sequence of states (Q) which are obtained by a successive transitions from the initial state (q_0) following the transition relation δ. A run of Configuration is a possible interleaving of σ_j for all EMMs.

[Location Graph] Location Graph is a directed graph (N, E). N is a set of locations, and a location is where a client can occupy, that is, a client walks to move from a location to another location. 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 correspond to particular edges that explicitly represent where clients can move from and to. Location

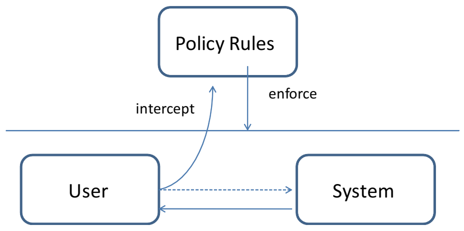

図 2 Two-tiered Framework

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 an actual place in the real world, but represents an abstract place corresponding to a set of places. It is abstract in that client behavior from any of the places in the set cannot be distinguished.

[**Location Projection**] Location Projection ξ is a function to extract a pair of client Id and its Location from state Q. Client is assumed here to be represented by an EMM; $\xi : Q \rightarrow$ ClientId $\times$ N. Note that ξ can work only on Client EMMs. 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 $\xi(\sigma_j)$.

[**Policy Rule**] Policy Rule monitors clients movement and enforces them to do as anticipated. It keeps track of Location Trail, and it checks whether a client trial movement 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 the location trail respectively. Currently, three forms of actions are considered; (a) permission rule to allow the user as he wants to do, (b) stop rule to instruct the user not to take any action, and (c) coercion rule to make the user to move as the policy rule calculates.

[**Policy Enforcer**] Policy Enforcer is an executing entity who enforces a given set of Policy Rules. Policy Enforcer is represented as a special kind of EMM. Since it uses Location Trail and checks the clients trial, a new architectural mechanism is needed to monitor all the clients movement, Two-tiered Framework. As seen in Figure 2, user movement trial is conceptually intercepted and checked with Policy Rule, and then its result is enforced.

4. Rewriting Logic Approach

4. 1 Overview of the Approach

Rewriting logic [2] is proposed by Jose Meseguer, which follows the tradition of algebraic specification methods. The logic extends order-sorted algebra to provide means to describe state changes. Languages such as Maude and CafeOBJ, based on Rewriting logic, are powerful enough to describe concurrent, non-deterministic systems. Such systems can be symbolically represented with appropriate level of abstractions in the languages. Furthermore, Maude tool provides advanced means to analyzing properties of the systems with various state-space search methods such as bounded reachability and LTL model-checking.

With Maude, the artifacts that we are interested in are modeled as a collection of entities executing concurrently. Their functional behavior is encoded with rewriting rules. As shown in Figure 2, the proposed framework achieves separating policy rules and behaviors of both the user and system. A bottom-line feature of the framework is to intercept the events occurred at the bottom layer, and the policy rules which will generate further events to enforce "policies" on the user. Since all of them can be represented as concurrently executing entities, the framework is ready to be encoded in Maude.

In this section we will show the example of using the algebraic logic approach to describe the guiding system. The following subsections explain the design and its Maude implementation in detail. We breakdown the guiding system as introduced in Section 2.. The foundation is the Location Graph and Extended Mealy Machine. Each entity in the guiding system acts as a state machine. Building on top of them is the guiding system.

4. 2 Location Graph

Location graph creates an abstract image of the physical location and the user machine can navigate through it. Figure 3 is an instance of location graph to represent an abstract view of the example in Figure 1. Users start from Location 0 and move along the arrows. At each location, Users may take some actions and they proceed to the next location if prescribed conditions are satisfied.

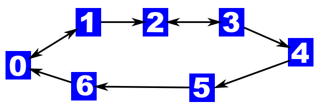

図 3 Using Directed Graph to Represent Guiding System

In encoding the location graph in Maude, we define several key Maude modules. Their importing relationships are shown in Figure 4. Each location has a maximum number of users to occupy it (Capacity). As users move around, their locations are changed, which should always be kept track of. We have defined Memory in addition to the location graph instance (ROOM1) that represents the static graph structure as shown in Figure 3.

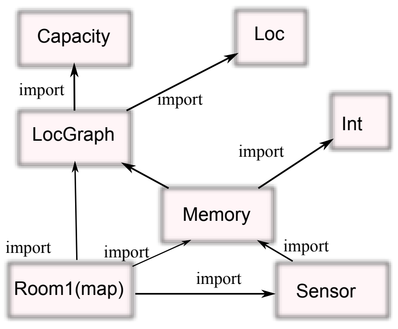

図 4　Modules for Location Graph

4. 3　Extended Mealy Machine

Extended Mealy Machine provides means to express dynamic behavior. We have followed the standard recipe, the Soup approach, to represent *state-machines* in Maude [2]. Figure 5 presents the modules to encode the machine. Soup is a general structural entity to have Machines and Events as components.

Each machine has a unique Id, the name of its type, and a set of attribute-value pairs. The values can be integer, unique Id, list of machine Ids or locations. The change of machine states is triggered by events that contain its Id as the receiver and satisfy specified conditions.

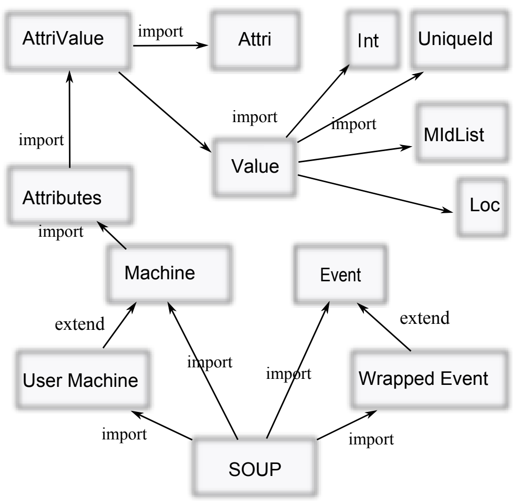

図 5　Modules for Extended Mealy Machine

As discussed in Section 2., some of the user behavior needs to be monitored by the policy before being processed by the system. As can be seen in Figure 2, some of the events generated by the user, which may represent the monitored behavior, is intercepted and dealt with by Policy Rules. A certain *tag* is needed to mark these event to show that they are monitored. We define the Wrapped Event module to serve this purpose.

When a machine representing the user (User Machine in Figure 5) generates a monitored event, it takes a form of a Wrapped Event. This wrapped event does not trigger state changes as others do, but is interpreted by the Policy. Then, Policy, by applying policy rules, decides what to do. In normal cases, the unwapped event is delegated to the machine that was supposed to receive it (Figure 6).

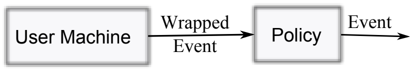

図 6　User Machine and Policy

4. 4　Guiding System

The last but not the least, it comes the module of the guiding system. We import both the Location Graph and Mealy Machine modules. There are six different types of modules in the guiding system and each of them is considered a EMM. Figure 7 shows the components and their interactions in the guiding system. The system includes components such as Door, GSys, CardReaders for either waiting or work room (Figure 1), and Plugins deciding different user behaviours. The instances of these components interact with each other and change its own internal states. The CardReader1 can access to DataBase to hold the profiles of all the registered users, examine user identities and unlock the door of waiting room. The CardReader2 extends CardReader1 so as to match a particular user to enter the work room. The GSys is the central controller of system state. It also takes care of notifying users to proceed. Plugins are adjacent to the GSys. They translate simple signal events into complex events such as user entering, exiting and arriving.

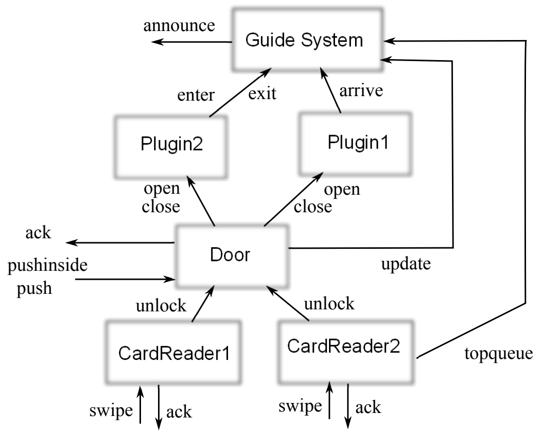

図 7　Guiding System Components and Their Interactions

One important part in this example is to design the GSys, the controller of the system. This module maintains the waiting list, sending *proceed* signal to the users and monitoring the entire system. The maintenance of the waiting list is interesting as we can use only three events to maintain the waiting queue and status of the work room which are,

- arrive: a user entering the waiting room from the outside which adds the user to the waiting list;
- enter: entering the work room from the waiting room which removes the user from waiting list and set the occupied sign true;
- exit: leaving the work room which releases the occupied sign.

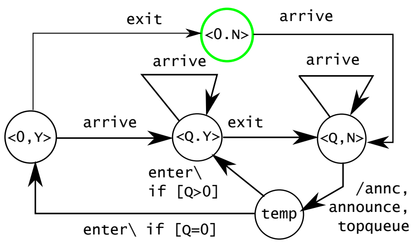

図 8 State Transition Diagram of GSys

The state changes are as shown in Figure 8. The user is put in the waiting queue when he *arrive*s, and removed from the queue when he *enter*s the work room. The work room is released when he *exit*s.

However, these user events are not detectable for the system. What the system can detect directly is only the information such as card readers being swiped, door being pushed, door status and sensor readings. We have to use these primitive events to infer the three events mentioned above. To do this we add PlugIn to the GSys. Figure 9 shows the plug-in that generates the enter and exit events.

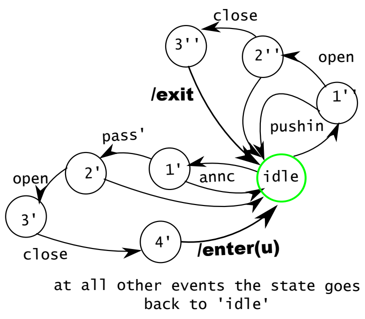

図 9 State Transition Diagram of Plugin for Work Room

4.5 Encoding in Details

After the design of modules, we show a few examples of how to encode the modules in the Maude.

The Machine module is a functional module (`fmod`) that provides a basic syntax for all the entities in the guiding system. We use the following code for a machine.

```
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 Maude module for the 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 rewriting rules represent the dynamic behavior as depicted by Figure 10.

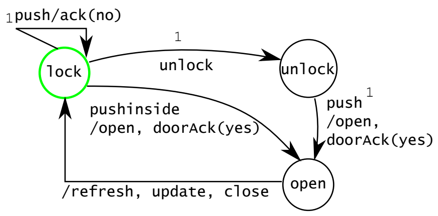

図 10 State Transition Diagram of Door

The state-transition diagram in Figure 10 shows its simple behavior, which is encoded by the rewriting rules below. Each rewriting rule corresponds to a transition in the diagram.

```
vars U : MachineId .
 rl [1] : push(D, U) < D : 'Door1 | 'lock ; null >
   => < D : 'Door1 | 'lock ; null >
      doorAck(U, false) .
 rl [2] : unlock(D) < D : 'Door1 | 'lock ; null >
   => < D : 'Door1 | 'unlock ; null > .
 rl [3] : push(D, U) < D : 'Door1 | 'unlock ; null >
   => < D : 'Door1 | 'open ; null >
      open('Plugin, D) doorAck(U, true) .
```

To open a door the user needs to send a *'push'* event. If the door is *locked*, the rule *[1]* applies. The door remains shut and send a *ack(fail)* event to the user. Rule *[2]* implies the door can be unlocked by the card reader. When the door is in *'unlock'* state, the state of Door is changed from unlocked to open if the door receives a *'push'*(push from inside) event from the user.

4.6 Validation in Maude

Tha Maude description developed so far is composed of 32 modules in about 800 lines of codes. About 60 % of codes are responsible for dynamic behavior, namely rewriting rules. Below present some results of analysis conducted in Maude 2.4 running on Panasonic CF-W7 under Windows/XP.

Firstly, the initial state of the Guiding System shown in Figure 1 is constructed. Secondly, some appropriate User(s) together with Policy Enforcer are added to the initial state. Here, we define the User so that it can initiate its action by receiving an initial event *start(u1)*. Furthermore, the Policy Enforcer is assumed to force the users to follow regular behavior; a user leaves Location 0 and returns to the same place after working in the Work Room. Such policy rules are easy to write because all the trials of movement are intercepted and are changed to regular behavior if abnormal. Below, the execution results are represented as traces of a tuple consisting of User and Location, (U, L).

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

```
('u1, 'L1), ('u1, 'L2), ('u2, 'L1), ('u1, 'L3),
('u2, 'L2), ('u1, 'L4), ('u1, 'L5), ('u1, 'L6),
('u2, 'L3), ('u1, 'L0), ('u2, 'L4), ('u2, 'L5),
('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 user u2 and another user 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 users 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), ('v1, 'L2), ('v1, 'L3),
('u2, 'L6), ('v1, 'L2), ('u2, 'L0)
```

Regardless of v1 behavior, the user u2 can move to L2 because more than one users can share the location. U2, however, has to wait before going to L3. The user v1 moves between L2 and L3, and only occasionally L3 becomes available.

Furthermore, *search* command can be used to check if there are execution paths from the initial state to target states with specified conditions. The following ensures that an execution path exists for u2 to enter the work room where *all* refers to the initial state of the Guiding System with a particular Policy Enforcer.

```
search all start(u2) =>* (REST:Soup)
  < 'u2 : 'User | 'L4 ; 'InWork ; (R1:Attributes) >
  < 'ichi : 'GSystem | 'busy ; (R:Attributes) >
```

5. Discussions and Conclusion

This paper first discussed the importance of separating policy rules from behavioral specifications of both the system and users. Then a two-tiered framework was proposed for such separation, and a way of encoding it in Maude, a specification language based on rewriting logic [2]. The method introduced a special tag to events that are to be intercepted by the Policy. An alternative encoding method may be possible where the notion of reflection in Maude is used.

Clear distinction between the system and its external environment has been recognized important [3]. It, however, does not consider separating the policy from the user, which together constitute the environment. Relationship between policy and system requirements is studied in security area [1]. As for the runtime enforcement of policies, Schneider [6] proposes security automata, a variant of Buchi automata. This paper uses policy rules which takes a form discussed in Section 3.. The rules basically look at an execution history as in the case of runtime monitoring [5].

文　献

[1] A.I. Anton, J.B. Earp, and R.A. Carter. Precluding Incongruous Behavior by Aligning Software Requirements with Security and Privacy Policies. Information and Software Technology, Vol. 45 (14), pages 967-977, 2003.

[2] M. Clavel et al. All About Maude – A High-Performance Logical Framework. Springer 2007.

[3] M. Jackson. Software Requirements & Specifications. Addison-Wesley 1995.

[4] M. Jackson. The Role of Formalism in Method. In Proc. FM'99, pages 56, 1999.

[5] S. Nakajima, K. Imai, and T. Tamai. Runtime Monitoring of Behavioral Specifications (in Japanese). IEICE SIGSS, October 2009.

[6] F.B. Schneider. Enforceable Security Policies. ACM Transactions on Information and System Security, Vol. 3, No.1, pages 30-50, 2000.