A Short Report for NII Intern

During the NII intern, I work with Professor Shin Nakajima whose research is in the area of algebraic model checking and program checking. I work on a project about separating the open system involving users and the policy rules that constraints the user behaviour. I need to implement the model of a guiding system using Maude, a algebraic specification environment and a workshop paper was submitted to a workshop in Kyoto. The technology used in this project includes algebraic specification, rewriting logic and state machine. In the rest of this short report, I will introduce what I learn from the project.

1 Two Tiered System

Defining precisely the environment in which computing systems are used is important at early stages of system development. 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 to constrain 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.

2 Two-tiered Framework

Policy Rule monitors clients movement and enforces them to do as anticipated. It keeps track of Location Trail,

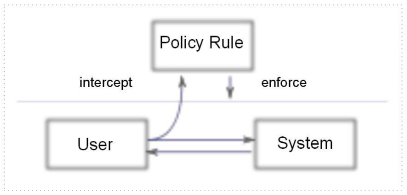


Figure 1. Two-tiered Framework

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 is an executing entity who enforces a given set of Policy Rules. A new architectural mechanism is needed to monitor all the user behavior, which is Two-tiered Framework. As seen in Figure 1, user behavior is conceptually intercepted and checked with Policy Rule, and then its result is enforced. The policy enforcer is one level above the system and the users. It is able to monitor everything in the lower level. The user behavior is sent up to the policy enforcer and it is sent back to lower level. As a result, the enforcer look over the system and make sure user behavior conform to the policy and the system is running properly. Policy rules include,

- Stop rule - to stop malicious user behavior. Ignore the incoming behavior and return nothing.

- Correction rule - to correct improper user behavior. Modify the incoming behavior and return the modified behavior.

- Allow rule - to allow proper user behavior. Return the behavior.

An example case following the proposed idea is encoded in Meseguer's rewriting logic so that various analysis, such as validation and property checking, is possible.

3 Maude, the environment for algebraic specifications

Maude is a high-performance reflective language and system. It has been influenced in important ways by the OBJ3 language, which can be regarded as an equational logic sublanguage. Maude can describe the models both statistics and dynamics. In other words, it supports both equational specification and programming and rewriting logic computation.

Maude uses algebraic specification to represent mathematical structures and functions. An algebraic specification achieves this goal by means of defining a number of sorts (data types, category of values) together with a collection of functions(operators) on them. The advantage of using Algebraic specification is its simplicity and strictness. While the disadvantage is it is relative hard to start with. In addition, the effect of modelling is not explicit. A module is a set of sorts and functions. The module that only have the structures and functions is called a 'Function Module'.

Rewriting logic is a logic of concurrent change that can naturally deal with state and with concurrent computations. It has good properties as a general semantic framework for giving executable semantics to a wide range of languages and models of concurrency. In particular, it supports very well concurrent object-oriented computation. The same reasons making rewriting logic a good semantic framework make it also a good logical framework, that is, a metalogic in which many other logics can be naturally represented and executed. The module that include such dynamic information is called a 'System Module'.

Besides the system and function module, Maude also support object oriented module.

Maude system includes two parts, Core Maude and Full Maude. Core Maude provides the basic functionality of Maude, including functional module, system module and some useful analysing tools. Full Maude extends Core Maude to include various analysing tool and modules such as object oriented module. Full Maude is written in Maude, and is thus platform-independent.

3.1 Function Module

Function Module represents the static structural and functions. It defines a number of sorts and functions, i.e. operators. Sort represent a category of values. Maude use 'subsort' keyword to specify the relation between sorts. The operators can be either constructor, i.e. fundamental operation that defines the basics of the algebra, or additional functions. Constant operations are usually constructors. Equations are used to represent the rule to simplify an expression. Expression with only constructors can not be simplified. Thus it is important to define how to represent a additional operator using constructors. Maude also support conditional equation. In addition current version of Maude2 can use pattern-match as condition too.

We can import definitions from other modules. Maude provide three types of importing.

- protecting - does not change the content of the imported module
- extending - can add new constructors to the imported module
- including - can add new constructors and redefining existing terms

Here we show an example of a function module.

```
fmod MACHINE is
sort Machine MachineId .
protecting QID .
subsort Qid < MachineId .

op Terminate : -> Machine [ctor] .
op <_> : MachineId -> Machine [ctor] .
op id_ : Machine -> MachineId .

var X : MachineId .
var Y : Machine .
eq id < X > = X .
ceq < id Y > = Y .

endfm
```

This example firstly define the name of the module. Inside MACHINE module, there are two sorts, Machine and MachineId. We define the relation with other module, which is importing QID without changing the content of QID. We also define the subsort relation between sorts. Besides supporting equational specification and programming, Maude also supports rewriting logic computation. In the second half, we have several operators. Terminate is a constant. ¡¿ is the constructor and id is additional operator. The equations then represent how to simplify the expressions.

3.2 System Module

System module can include everything that is in a function module. In addition, it can also represent the transition that occurs within or between structures, i.e. rewriting logic.

Rewriting laws and equations solve very different problems. Equations are better than rewrite laws at simplification; while rewrite laws are better at expressing problems with just one level of simplicity. Rewriting laws also can illustrate constitutional changes that equations canft; though an equation may simplify an expression, the expression is still mathematically equal to its predecessor.

In the case of more than one rewriting rules share the same left hand side, Maude need to decide which rule to be executed. The selection can be either deterministic or non-deterministic. The rewrite command is a default strategy provided by Maude. It always choose the first rule. To use other ordering strategies, Maude provide fair rewrite, which guarantee using all the rules.

Conditional Rewrite Logic are supported too. Following is an example of system module.

```
mod STATEMACHINE is
protecting MACHINE .
protecting EVENT .
including SOUP .

var X : MachineId .
var Y : EventId .
rl [rule1] : evt '3' ('0')  < '0' > => < '1' >  .
crl [rule3] : evt Y ( X )   < X > => Terminate if X == Y  .

endm
```

Besides the same information that shows in a function module, system module STATEMACHINE also defines two rewriting rules. The first one represents when both evt'3'('0') and <'1'> appears, they should transform to <'1'>. The second rule is a conditional rule, which means when Y equals X, the next state will be Termination.

3.3 Analyse The Models

There are three tools to analyse the rewriting, rewriting, search and model checking. Rewriting executes the model in the depth first order. It trace a specific execution path until no more rules fits. Search executes in width first order. It searches all the possible paths start from current state and looking for a match to the destination state. Maude's model checking uses LTL. It also uses pattern matching.

4 Model a Guiding System use algebraic specifications

4.1 Introduction to the Guiding System

The Guiding System is a user-navigation system for clients to access particular resources privately. The guiding system consists of three areas, outside, waiting room and workrooms. The waiting room only allows clients who is registered customer of the system while one workroom only allows the property owner to enter. Clients arriving at the system check at the waiting room door firstly. Those pass the check enter and then wait in the waiting room. They wait until being notified enter the workroom and use their property. The clients leave the system afterwards.

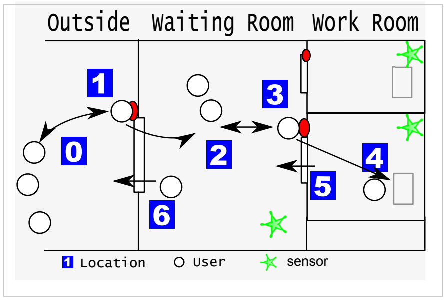


Figure 2. A Guiding System Example

As in Figure2, the numbers represent areas, such as outside, in front of a card reader, in front of the door, so on and so forth. The clients can move in the direction of the arrow.

Besides the above regular scenario, many unexpected cases could happen. For example, a client might try to go to the work room before he is called, a client might walk around in the waiting room disturbing others by blocking the door to the work room and some might be absent when he is called and would not use the resource at all. The guiding system could suffer from unexpected user behaviour. It needs a set of rules to regulate clients behaviour. For this reason, the guiding system serves a good example for the two-tiered framework.

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 wireless communications. It, however, would results 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 behaviour. 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." In such design, the system can be defined without being over-engineered since the clients are assumed to follow the regular scenarios. The implicit assumptions on the client behaviour are made explicit in the form of policy rules.

The second method is used in this project. The policy rules are considered as a separate layer of a user involved system. The user behaviours are sent to undergo the policy check to enforce the legibility.

4.2 Breakdown the guiding system

We breakdown the guiding system as the Location Graph. Extended Mealy Machine and system modules building on top of them is the guiding system. Each entity in the guiding system acts as a state machine.

4.2.1 Location Graph

Location graph creates an abstract image of the physical location. the user machine can navigate through it.

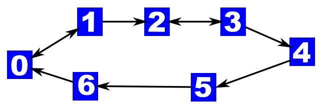


Figure 3. Using Directed Graph to Represent Guiding System

Figure 17 is an instance of location graph to represent an abstract view of the example in Figure 2. 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.

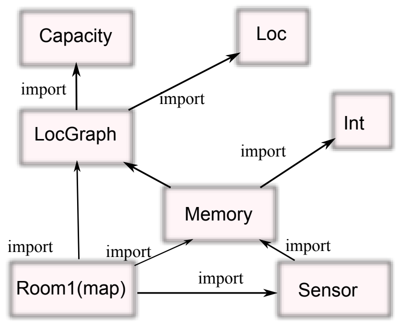


Figure 4. Modules for Location Graph

In encoding the location graph in Maude, we define several modules. Their importing relationships are shown in Figure 4. Each location has a unique Id (Qid) a maximum number of users it can holds (Capacity). LocGraph defines operators that navigates the locations. As users move around, their locations are changed. The system must always be aware of user location. We have defined Memory to store the status of the graph. In addition, ROOM1 extends Memory and LocGraph. It stores the connectivity and the capacity of the locations.

4.2.2 Extended Mealy Machine

The guiding system is comprised of many entities which interact with each other. In the system a user autonomous to move between locations. We also need a hypothetical component to enforce the policy rules executes independently. Such observation leads us to adapting a modeling method of using a set of communicating state-transition machines.

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.

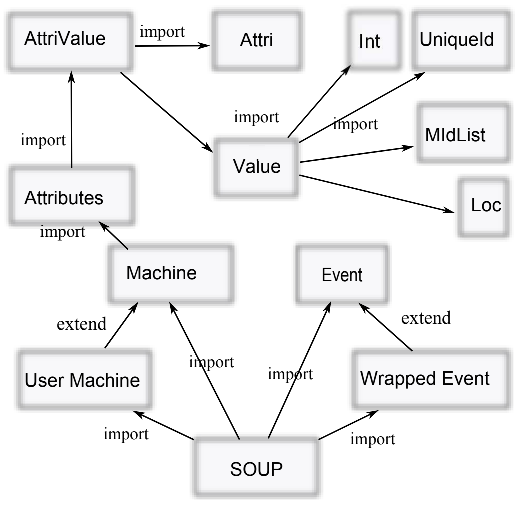


Figure 5. Modules for Extended Mealy Machine

Figure 5 presents the modules to encode the machine. Each machine has a unique Id, the name of its type, and a set of attribute-value pairs. The values of an attribute 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. Soup is a general structural entity to have Machines and Events components. We also include two more components in the SOUP, which is User Machine and Wrapped Event.

We have defined the User Machine in addition to normal Machine. Here we take advantage of the Maude features for pattern matching. Having a separate property for location is faster to match than having it in the attributes. As we can see, the content of the two modules are as following:

Machine
$$< MachineId : TypeId | StateId ; Attributes >$$
vs
User Machine
$$< MachineId : TypeId | Loc ; StateId ; Attributes >$$

In the favor of the speed, we introduced the User Machine.

As discussed in the first Section, some of the user behavior needs to be monitored by the policy before being processed by the system. As can be seen in Figure 1, some of the events generated by the user, which represent the monitored behavior, are intercepted and dealt with by Policy Rules. A *tag* is needed to mark these event to show that they are monitored. We define the Wrapped Event in addition to Extended Mealy Machine to serve this purpose.

When a machine representing the user (User Machine in Figure5) generates a monitored event, it takes the 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.

4.2.3 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.

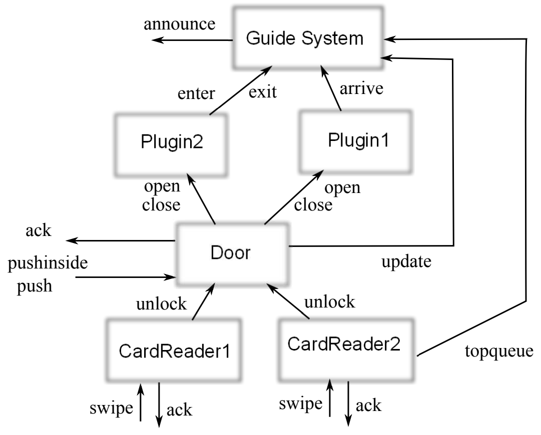


Figure 6. Guiding System Components and Their Interactions

Figure 6 shows the components and their interactions in the guiding system. Controller and Plugins are in control of the system. The components such as Door, CardReaders which are responsible for interacting with User machine and sending basic signals to the Controller and its Plugins.

4.2.4 Card Readers and Door

This sections shows the design of the state transition diagram for each components.

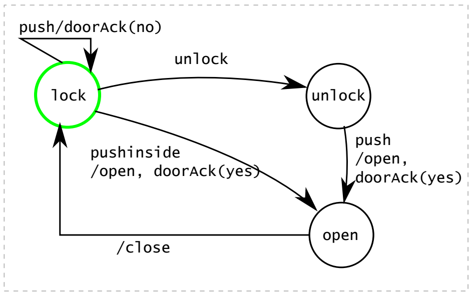


Figure 7. State Transition Diagram of Door

The initial state of a door is *lock*, in this state, the user can not open the door from outside, but they can open the door from inside. If the user passes the id check, the card reader sends a *unlock* event to the door, so that the door is *unlock*ed. At this time, the user can *push* the door and enter. Then the door closes automatically. During the process, the door will send a *open* and a *close* event to the corresponding plugin. The door also sends acknowledgement signal to the users as feedback.

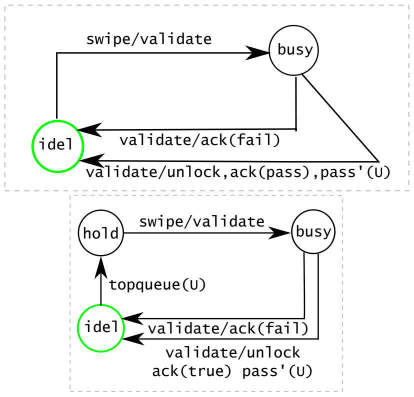


Figure 8. State Transition Diagram of CardReaders

The CardReader1 hold the profiles of all the registered users, examine user identities and let the register clients enter. The CardReader2 extends CardReader1 so as to match a particular user to enter the specified work room. Both readers(Figure8) are in *idle* status initially. Card Reader 2 will receive the customer id from *topqueue* event and change to *hold* state. A *swipe* event will make both transit to *busy* state and begin validating the request. The card readers send feedbacks to the user. Card readers send a *pass* event to the corresponding plugin if the user passes the check as well as the event to unlock the corresponding door.

4.2.5 Guiding System and its plugins

The Controller is the central controller of system state. It maintains the wait list and keep track of work room status. It notifies the users on top of the wait list to proceed. The maintenance of the wait list is interesting as we are able to use three events to maintain the wait list and status of the work room, which are,

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

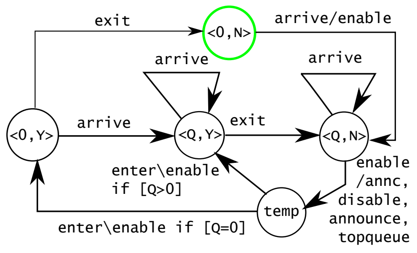


Figure 9. State Transition Diagram of Controller

The state changes are as shown in Figure 9. The user is put in the wait list 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, we still have a problem. The three 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 and door status. We have to use these primitive events to infer the three events mentioned above. To do this we add Plugin modules. Plugins are adjacent to the Controller. They translate simple signal events into complex events such as user entering, exiting and arriving. The following figures shows what happens in the system should these complex event occur.

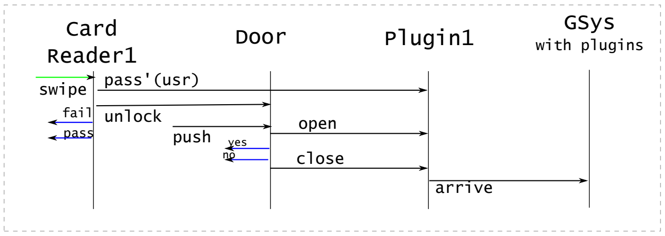


Figure 10. Arrival Event

An arrival event is happened when pass from waiting room door card reader, open from waiting room door and close from waiting room door happens successively.

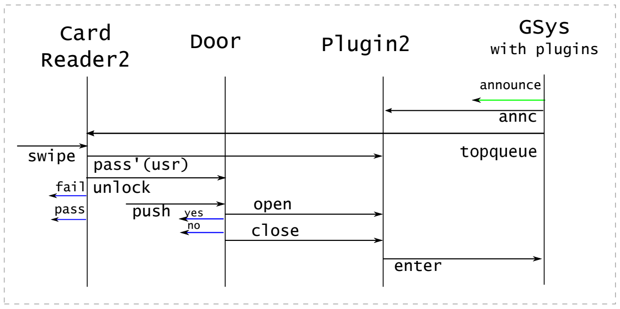


Figure 11. Enter Event

An arrival event is happened when annc from Controller, pass from work room door card reader, open from work room door and close from work room door happens successively.

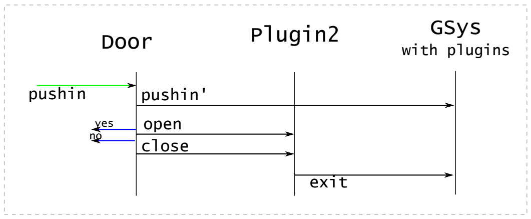


Figure 12. Exit Event

An exist event is happened when pushin from work room door, open from work room door and close from work room door happens successively.

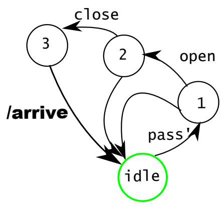


Figure 13. State Transition Diagram of Plugin for Work Room

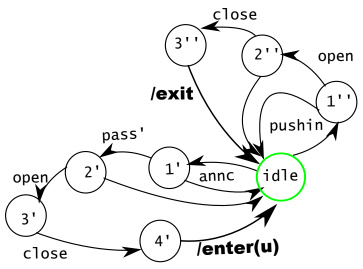


Figure 14. State Transition Diagram of Plugin for Work Room

Figure 13 and Figure 14 show the plug-in state transition diagram that generates the enter and exit events.

4.2.6 User Machine

The User Machine is the generator of user events. It allows the system to execute autonomous.

The events that generated by the users trigger changes of entity status. push change the door state from unlock to open; pushin change the door state from lock to open; swipe changes card reader state from idle to work.

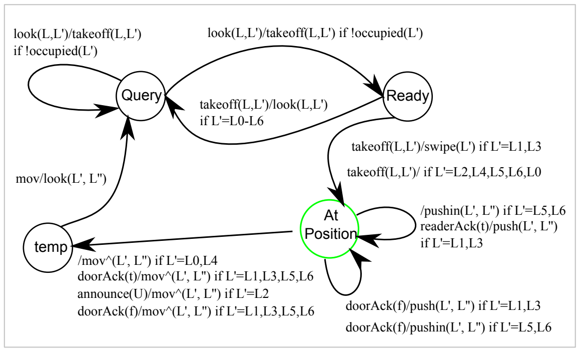

look(L,L')/takeoff(L,L')
if !occupied(L')
look(L,L')/takeoff(L,L') if !occupied(L')
Query
Ready
takeoff(L,L')/look(L,L')
if L'=L0-L6
takeoff(L,L')/swipe(L') if L'=L1,L3
takeoff(L,L')/ if L'=L2,L4,L5,L6,L0
mov/look(L', L")
At
Position
/pushin(L', L") if L'=L5,L6
readerAck(t)/push(L', L")
if L'=L1,L3
temp
/mov^(L', L") if L'=L0,L4
doorAck(t)/mov^(L', L") if L'=L1,L3,L5,L6
announce(U)/mov^(L', L") if L'=L2
doorAck(f)/mov^(L', L") if L'=L1,L3,L5,L6
doorAck(f)/push(L', L") if L'=L1,L3
doorAck(f)/pushin(L', L") if L'=L5,L6

Figure 15. State Transition Diagram User Machine

Except the event to interact with system components. There are basic events that allow the User Machine move autonomous in the location graph.

- look to start checking the availability of next location;
- takeoff to get prepared to move;
- step represents the user movement. It is wrapped and sent to the PolicyEnforcer The user has some basic behaviours such as,

As shown in Figure.**??**, the User Machine starts from atPosition. At a position, the user machine can carry out some actions. Then the user machine intend to move to the next location, so he sends a step event which is wrapped and examined by the Policy enforcer. If the policy enforcer agrees with it. The step event is unwrapped and the user machine goes to look if the next location is available. If so, he take off and prepare to move. Once he move to the next location, he is in atPosition state again. During the process, the user can also choose to not follow the normal procedure. He can choose to stay at the current position instead of moving forward, and he can also choose keep checking the next location status instead of moving.

4.2.7 Policy Enforcer and Exceptional Behaviour

As in our example, the system only care about the user behaviours that affects the system status, that is the number of clients in each node in the location graph. The policy module is designed to monitor the events that changes the system status, instead of all the user generated events. Therefore the user behaviour such as swipe and push are not interesting, and only the action of step to the next location in the graph is wrapped and sent to the Policy Enforcer. PolicyEnforcer monitors the status of the system and decide if UserMachine's wrapped step event is appropriate. As we discussed in the previous section, the Policy Enforcer examine the user wrapped event and return proper event. But how should Policy work?

Let's consider some exceptional behaviours in this section. For example, UserMachine moves constantly between two locations 2 and 3 in the location graph(Figure**??**) after he is notified. He is on top of the wait list and the system can not remove him from the list until he enters the workroom. Therefore every user behind him has to wait and the system is blocked. A live lock occurs in the system. If UserMachine generates such events, Policy Enforcer need 'stop' it.

In this situation, the policy enforcer detects the problem when the counter exceeds the limit. Then the policy enforcer try to resolve it. The solution to a guiding system will be: Notify system manager(physically) and Remove the User from the queue(electronically).

4.3 Encoding in Maude

After the design of modules, we show a few examples of how to encode the modules in the Maude. Please refer to the Appendix for the full code.

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 7.

The state-transition diagram in Figure 7 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 >
```

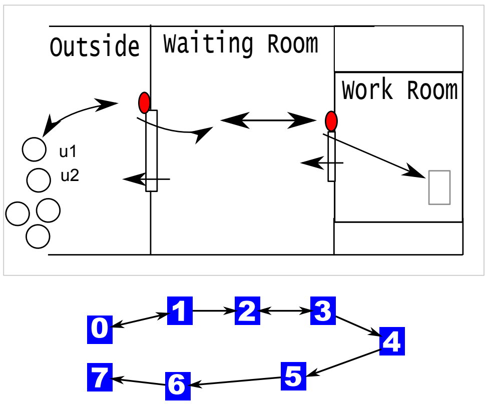
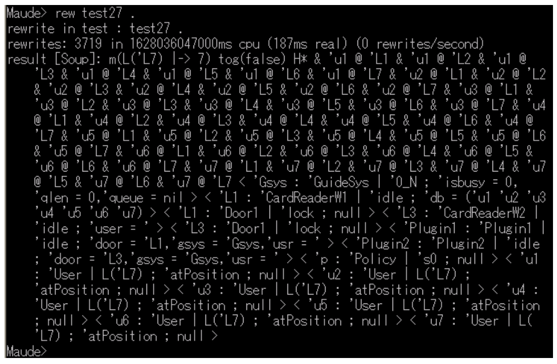


Figure 16. A Guiding System Experiment



```
Maude> rew test27 .
rewrite in test : test27 .
rewrites: 3719 in 1628036047000ms cpu (187ms real) (0 rewrites/second)
result [Soup]: m(L('L7) |-> 7) tog(false) H* & 'u1 @ 'L1 & 'u1 @ 'L2 & 'u1 @
    'L3 & 'u1 @ 'L4 & 'u1 @ 'L5 & 'u1 @ 'L6 & 'u1 @ 'L7 & 'u2 @ 'L1 & 'u2 @ 'L2
    & 'u2 @ 'L3 & 'u2 @ 'L4 & 'u2 @ 'L5 & 'u2 @ 'L6 & 'u2 @ 'L7 & 'u3 @ 'L1 &
    'u3 @ 'L2 & 'u3 @ 'L3 & 'u3 @ 'L4 & 'u3 @ 'L5 & 'u3 @ 'L6 & 'u3 @ 'L7 & 'u4
    @ 'L1 & 'u4 @ 'L2 & 'u4 @ 'L3 & 'u4 @ 'L4 & 'u4 @ 'L5 & 'u4 @ 'L6 & 'u4 @
    'L7 & 'u5 @ 'L1 & 'u5 @ 'L2 & 'u5 @ 'L3 & 'u5 @ 'L4 & 'u5 @ 'L5 & 'u5 @ 'L6
    & 'u5 @ 'L7 & 'u6 @ 'L1 & 'u6 @ 'L2 & 'u6 @ 'L3 & 'u6 @ 'L4 & 'u6 @ 'L5 &
    'u6 @ 'L6 & 'u6 @ 'L7 & 'u7 @ 'L1 & 'u7 @ 'L2 & 'u7 @ 'L3 & 'u7 @ 'L4 & 'u7
    @ 'L5 & 'u7 @ 'L6 & 'u7 @ 'L7 < 'Gsys : 'GuideSys | 'O_N ; 'isbusy = 0,
    'qlen = 0,'queue = nil > < 'L1 : 'CardReaderW1 | 'idle ; 'db = ('u1 'u2 'u3
    'u4 'u5 'u6 'u7) > < 'L1 : 'Door1 | 'lock ; null > < 'L3 : 'CardReaderW2 |
    'idle ; 'user = ' > < 'L3 : 'Door1 | 'lock ; null > < 'Plugin1 : 'Plugin1 |
    'idle ; 'door = 'L1,'gsys = 'Gsys,'usr = ' > < 'Plugin2 : 'Plugin2 | 'idle
    ; 'door = 'L3,'gsys = 'Gsys,'usr = ' > < 'p : 'Policy | 's0 ; null > < 'u1
    : 'User | L('L7) ; 'atPosition ; null > < 'u2 : 'User | L('L7) ;
    'atPosition ; null > < 'u3 : 'User | L('L7) ; 'atPosition ; null > < 'u4 :
    'User | L('L7) ; 'atPosition ; null > < 'u5 : 'User | L('L7) ; 'atPosition
    ; null > < 'u6 : 'User | L('L7) ; 'atPosition ; null > < 'u7 : 'User | L(
    'L7) ; 'atPosition ; null >
Maude>
```

Figure 17. A Guiding System Experiment Result Screenshot

```
  => < 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.

The system allows some degree of uncertainty to illustrate the situations in the real world. The non-deterministic rules, for example in the user machine, is expressed in two rules with same left hand side and conditions but different right hand side. To enable the non-deterministic, instead of using rewrite command which always execute the rules with highest priority, we use Maude fair rewrite command to make sure all the rules are executed.

```
 rl [r31] :
 takeoff(M,X') < M : 'User | X ; 'ready ;  R > W
 => < M : 'User | X' ; 'query ; R >  .
 crl [r32] :
 takeoff(M,X') < M : 'User | X ; 'ready ;  R > W
 => < M : 'User | X' ; 'atPosition ; R >
   if X' =/= 'L1 and X' =/= 'L3 .
```

For the other parts of the code, please refer to the appendix.

5 Result and Conclusion

Having a complete system, we can carry out some experiments to show how the system works. More test cases can be found in setup.maude. Here we show one of the experiment.

```
Test27 =
makeUser('u2) makeUser('u1) makeUser('u3) makeUser('u7)
makeUser('u4) makeUser('u6) makeUser('u5) m(inc(inc
(inc(inc(inc(inc(inc(nobody, L('L0)), L('L0)), L('L0)),
L('L0)), L('L0)), L('L0)), L('L0)))
makeDoor('L1) makeCardReaderW1('L1, ('u1 'u2 'u3 'u4
'u5 'u6 'u7)) makeDoor('L3) makeCardReaderW2('L3)
makePlugin1('Plugin1, 'Gsys, 'L1)
makePlugin2('Plugin2, 'Gsys, 'L3)
makeGSys('Gsys) tog(false)
makePolicy('p)  H*
```

This setup consists of 1 waiting room, 1 work room, corresponding doors and card readers and the controller. There are 7 users queuing to use the work room. We use fair rewrite command to analyse the model. H* is used to trace the route of each user.

The result is as such,

6 Future Work

The project discusses how the policy rules play an essential role in open systems that involves users. It proposes a framework for system description in which user behavior and policy rules are explicitly separated. We also implement an guiding system example to demonstrate this idea. As this is the initial step of the research, there are many things can be done in the future.

Investigate the policy for different application

The systems involving users are widely used in the modern world. They are used different domain, serve different purpose and have different emphasis. The example we use in this project, the guiding system is just the top of an iceberg. Therefore, it is interesting to discover more about the policies for different domain. It is also interesting to optimise the number of rules and their effects.

Develop the current example

Develop the current project to simulate more complex, more real situations would be an interesting choice too. It includes adding more non-deterministic and constructing system with more locations and more complex edges.

Make use more Maude analysis functionality

We haven't use the search and model checking technique provided by Maude yet. Using these technology will help us to find more interesting situation.

Follow Up Researches in Software Engineering

This project is the initial step of the research in software engineering. The idea of using stand alone policies, despite using the example of a real world application, could inspire software engineering technology. For example, the policy we used in this example to monitor the number of repeat movement can, for sure, be used to resolve live lock.

It is worthy to note that the policy in this project is similar to the concerns in aspect oriented design, whereas this project accomplishes it in a different way.

Appendix: Maude code

```
--- LocGraph.maude
--- Basic Definitions for Location Graph
--- either static or dynamic, the latter being a kind of cache
---

fmod LOCATION is
  sort Loc .
  protecting QID .
  op L : Qid -> Loc [ ctor ] .
endfm

view Loc from TRIV to LOCATION is ---We use views to specify how a particular target module or theory is claimed to sa
  sort Elt to Loc .
endv

fmod CAPACITY is
  sort Capacity .
  protecting INT .
  op c : Int -> Capacity [ ctor ] .
  op many : -> Capacity [ ctor ] .
  op _>_ : Capacity Int -> Bool .

  vars N, M : Int . var C : Capacity .
  eq many > M = true .
  eq c(N) > M = N > M .
endfm



fmod LOC-GRAPH is
  protecting LIST{Loc} .
  protecting CAPACITY .
  op next : Loc -> List{Loc} .
  op max : Loc -> Capacity .
  op allowed : Loc Int -> Bool .
  op choose : List{Loc} -> Loc .

  var X : Loc . var N : Int .
  eq allowed(X,N) = max(X) > N .
  op D : Qid -> Qid .
  var XL : List{Loc} .
  eq choose(X) = X .
  eq choose(X XL) = X .
endfm

  --- owise: acts like a conditional equation whose condition
  ---is the exact opposite of the first equation.


fmod MEMORY is
  sort Memory .                --- should be a subsort of Soup
  protecting LOC-GRAPH .
  protecting ARRAY{Loc, Int0} .

  op m : Array{Loc, Int0} -> Memory [ ctor ] .
  op nobody : -> Array{Loc, Int0} [ ctor ] .
  op get : Memory Loc -> Int .
  op inc : Array{Loc,Int0} Loc -> Array{Loc,Int0} .
  op dec : Array{Loc,Int0} Loc -> Array{Loc,Int0} .
  op move : Memory Loc Loc -> Memory .
  op space? : Memory Loc -> Bool .

  var M : Array{Loc,Int0} . vars X Y : Loc .
  eq  get(m(M),X) = (M)[X] .
  eq  inc(M,X) = insert(X,(M)[X] + 1, M) .
  eq  dec(M,X) = insert(X,(M)[X] - 1, M) .

  eq move(m(M),X,Y) = m(inc(dec(M,X),Y)) .
  eq space?(m(M),X) = allowed(X,(M)[X]) .
endfm

---

fmod SENSOR is
  sort Sensor .
  protecting MEMORY .
  op SensorB : Qid Memory -> Bool .
  op SensorI : Qid Memory -> Int .

  var X : Qid . var M : Memory .
  eq SensorB(X,M) = SensorI(X,M) > 0 .
endfm

--- Concrete Data

fmod ROOM1 is
  including LOC-GRAPH .
  including MEMORY .
  including SENSOR .

  eq next(L('L0)) = L('L1) .
  eq next(L('L1)) = L('L2) .
  eq next(L('L2)) = L('L3) .
  eq next(L('L3)) = L('L4) .
  eq next(L('L4)) = L('L5) .
  eq next(L('L5)) = L('L6) .
  eq next(L('L6)) = L('L7) .

  ---eq L('L7) = L('L0) .
  eq next(L('L2)) = L('L6) .
  eq next(L('L3)) = L('L2) .

  eq max(L('L0)) = many .
  eq max(L('L1)) = c(1) .
  eq max(L('L2)) = many .
  eq max(L('L3)) = c(1) .
  eq max(L('L4)) = c(1) .
  eq max(L('L5)) = c(1) .
  eq max(L('L6)) = many .
  eq max(L('L7)) = many .

  eq D('L5) = 'L3 .
  eq D('L6) = 'L1 .

  eq nobody = L('L0) |-> 0 .

  var M : Memory .
  eq SensorI('Waiting,M) = get(M,L('L2)) + get(M,L('L3)) + get(M,L('L6)) .
  eq SensorI('Work1,M)    = get(M,L('L4)) + get(M,L('L5)) . --- work room No1.

  --- here the sensor id should conform to the map and
  --- the id of rooms doors and sensors in statemachine.maude

endfm




--- State Machine ====

fmod MID is
  sort MachineId .
  protecting QID .  subsort Qid < MachineId .
endfm

view MachineId from TRIV to MID is
  sort Elt to MachineId .
endv

fmod MID-LIST is
```

```
  sort MIdList .
  protecting LIST{MachineId} .
  subsort List{MachineId} < MIdList .
endfm

fmod VALUE is
  sort Value .
  protecting QID .  protecting INT . protecting LOC-GRAPH . protecting MID-LIST .
  subsorts  Qid Int Loc MIdList < Value .

endfm




fmod EVENT is
  sort Event .
  protecting MID . protecting VALUE .


  --- send to door
  ops push pushin : MachineId MachineId -> Event .
  op unlock : MachineId -> Event .

  ---- send to card reader waiting room
  op swipe : MachineId MachineId -> Event .

  --- send to card reader workroom
  op topqueue : MachineId MachineId -> Event .

  ----send to Plugin
  op annc : MachineId MachineId -> Event .
  ops open close : MachineId MachineId -> Event .
  ---op update : MachineId MachineId -> Event .
  op pass' : MachineId MachineId MachineId -> Event .
  op pushin' : MachineId MachineId -> Event .

  ---send to Gsys
  ops arrive enter : MachineId MachineId MachineId -> Event .
  op exit : MachineId MachineId -> Event .


  ---send to user
  op announce : MachineId MachineId -> Event .
  ops readerAck doorAck  : MachineId Bool -> Event .

  --- interesting user event
  op step : MachineId StateId Attributes Loc Loc -> Event .


endfm

fmod WRAPPER is
  sort WREvent .
  extending EVENT .   subsort WREvent < Event .
  op ^_(_) : Event MachineId -> WREvent [ ctor ] .
  op ^_    : Event -> WREvent [ ctor ] .
  op unwrap : WREvent -> Event .

  var E : Event . var M : MachineId .
  eq unwrap(^(E)(M)) = E .
  eq unwrap(^(E))    = E .
endfm


fmod ATTRIBUTE is
  sort Attributes .
  protecting VALUE .
  sort AttrId .  subsort Qid < AttrId .
  sort AttrValue .   subsort AttrValue < Attributes .

  op null : -> AttrValue [ ctor ] .
  op _=_ : AttrId Value -> AttrValue [ ctor prec 20 ] .
  op _,_  : Attributes Attributes -> Attributes [ ctor assoc comm id: null prec 25 ] .

  --- the precedence of , and = need to be resolved. before that please use brakets ()
endfm

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

fmod USER-MACHINE is
  including MACHINE .
  sort User . subsort User < Machine .
  op <_:_|_;_;_> : MachineId TypeId Loc StateId Attributes -> User [ ctor ] .
endfm


mod SOUP is
  sort Soup .
  protecting WRAPPER .
  protecting USER-MACHINE .  protecting MEMORY . protecting HISTORY .
  subsorts History Memory Machine Event < Soup .
  op empty : -> Soup [ ctor ] .
  op __ : Soup Soup -> Soup [ ctor assoc comm id: empty ] .
endm




---- Guiding System
--- in LocGraph.maude
--- in StateMachine.maude



mod DOOR1 is
  extending SOUP .

  op makeDoor : MachineId -> Machine [ctor] .
  var D : MachineId .
  eq makeDoor(D) = < D : 'Door1 | 'lock ; null > . --- S = 'Waiting or 'Work

  --- could  MachineId be removed since the event is to outside world but not to another module


  --- states: 'lock, 'unlock, 'open

  vars U : MachineId .  --- Bool : true/false

  --- the priority of the rules matters. be careful. how to set priority?
  --- when push and unlock at the same time, it seems the push always gets the advantage.

  rl [close] : < D : 'Door1 | 'open ; null >
                => < D : 'Door1 | 'lock ; null > close('Plugin, D) .
                --- The id of the door and corresponding sensor and the room should be the same

  rl [pushinside] : pushin(D, U) < D : 'Door1 | 'lock ; null >
                => < D : 'Door1 | 'open ; null > open('Plugin, D) doorAck(U, true)
  pushin'('Plugin, D) .

   rl [unlock] : unlock(D) < D : 'Door1 | 'lock ; null >
                => < D : 'Door1 | 'unlock ; null > .

   rl [pushopen] : push(D, U) < D : 'Door1 | 'unlock ; null >
                => < D : 'Door1 | 'open ; null > open('Plugin, D) doorAck(U, true) .
                --- should the Gsys MachineId be presented using a var instead of constant?

   rl [pushno] : push(D, U) < D : 'Door1 | 'lock ; null >
                => < D : 'Door1 | 'lock ; null > doorAck(U, false) .
```

```
endm


mod CARDREADERW1 is
  extending SOUP .
  op makeCardReaderW1 : MachineId MachineId -> Machine [ctor] .
  var C : MachineId . var DB : MIdList .
  eq makeCardReaderW1(C, DB) = < C : 'CardReaderW1 | 'idle ; 'db = DB > . --- C = 'Waiting or 'Work

  --- internal events
  op validate : MachineId MachineId -> Event . --- itself id, user id
  --- ops pass fail : MachineId MachineId -> Event . --- itself id, user id

  --- states: 'idle and 'busy

  var U : MachineId . var R : Attributes .

  rl [swipeact] : swipe(C, U) < C : 'CardReaderW1 | 'idle ; R >
                    => validate(C, U)  < C : 'CardReaderW1 | 'busy ; R > .

  --- how to define fail or pass? now it is un derterministic. This must be resolved.
  crl [validatef] : validate(C, U) < C : 'CardReaderW1 | 'busy ; 'db = DB , R >
                   => readerAck(U, false)  < C : 'CardReaderW1 | 'idle ; 'db = DB ,R >
   if occurs(U, DB) == false .


  crl [validatep] : validate(C, U) < C : 'CardReaderW1 | 'busy ; 'db = DB , R >
                    => readerAck(U, true) < C : 'CardReaderW1 | 'idle ; 'db = DB , R >
                    pass'('Plugin, U, C) unlock(C) if occurs(U, DB) == true .

endm




mod CARDREADERW2 is
  extending SOUP .
  extending CARDREADERW1 .
  op makeCardReaderW2 : MachineId -> Machine [ctor] .
  var C : MachineId .
  eq makeCardReaderW2(C) = < C : 'CardReaderW2 | 'idle ; 'user = ' > . --- C = 'Waiting or 'Work

  --- states: 'idle, 'await and 'busy

  var U U' : MachineId .

  rl [sysact] : topqueue(C, U) < C : 'CardReaderW2 | 'idle ; 'user = ' >
                => < C : 'CardReaderW2 | 'await ; 'user = U > .

  rl [swipeact] : swipe(C, U') < C : 'CardReaderW2 | 'await ; 'user = U >
                    => validate(C, U')  < C : 'CardReaderW2 | 'busy ; 'user = U > .

  --- how to define fail or pass? now it is un derterministic. This must be resolved.
   crl [validatef] : validate(C, U') < C : 'CardReaderW2 | 'busy ; 'user = U >
                => readerAck(U', false)  < C : 'CardReaderW2 | 'idle ; 'user = ' >  if U =/= U' .


  crl [validatep] : validate(C, U') < C : 'CardReaderW2 | 'busy ; 'user = U >
                    => readerAck(U, true)  < C : 'CardReaderW2 | 'idle ; 'user = ' >
                        pass'('Plugin, U, C) unlock(C)   if U == U' .
endm



mod PLUGIN1 is
  extending SOUP .

  op makePlugin1 : MachineId MachineId MachineId -> Machine [ctor] .

  vars P G D : MachineId .
  eq makePlugin1(P, G, D) = < P : 'Plugin1 | 'idle ; 'gsys = G , 'door = D, 'usr = qid("") > .
  --- states: 'idle, 'a1, 'a2, 'a3, 'b1, 'b2, 'b3, 'b4, 'c1, 'c2 and 'c3

  vars D1 U C : MachineId .
  var R : Attributes .
  var E : Event .

  ----

  crl [a1] : < P : 'Plugin1 | 'idle ; R , 'door = D, 'usr = ' > pass'('Plugin, U, D)
             => < P : 'Plugin1 | 'a1 ; R , 'door = D, 'usr = U > if D == 'L1 .
  ---&&

  rl [a2] :  < P : 'Plugin1 | 'a1 ; R , 'door = D > open('Plugin, D)
             => < P : 'Plugin1 | 'a2 ; R , 'door = D > .

  rl [a3] : < P : 'Plugin1 | 'a2 ; R , 'door = D > close('Plugin, D)
             => < P : 'Plugin1 | 'a3 ; R , 'door = D > .

  rl [a0] : < P : 'Plugin1 | 'a3 ; R , 'door = D , 'gsys = G , 'usr = U >
             => < P : 'Plugin1 | 'idle ; R, 'door = D , 'gsys = G , 'usr = ' > arrive(G, D, U) .

------
  crl [c1] : < P : 'Plugin1 | 'idle ; 'door = D , R > pushin'('Plugin, D)
             => < P : 'Plugin1 | 'c1 ; 'door = D , R > if D == 'L1 .

  rl [c2] : < P : 'Plugin1 | 'c1 ; R , 'door = D > open('Plugin, D)
             => < P : 'Plugin1 | 'c2 ; R , 'door = D >  .

  rl [c3] : < P : 'Plugin1 | 'c2 ; R , 'door = D > close('Plugin, D)
             => < P : 'Plugin1 | 'c3 ; R , 'door = D > .

  rl [c0] : < P : 'Plugin1 | 'c3 ; 'gsys = G , 'door = D, R >
             => < P : 'Plugin1 | 'idle ; 'gsys = G , 'door = D , R > .

endm


mod PLUGIN2 is
  extending SOUP .

  op makePlugin2 : MachineId MachineId MachineId -> Machine [ctor] .

  vars P G D : MachineId .
  eq makePlugin2(P, G, D) = < P : 'Plugin2 | 'idle ; 'gsys = G , 'door = D, 'usr = qid("") > .

  --- states: 'idle, 'b1, 'b2, 'b3, 'b4, 'c1, 'c2 and 'c3

  vars D1 U C : MachineId .
  var R : Attributes .
  var E : Event .


  crl [b1] : < P : 'Plugin2 | 'idle ; R , 'door = D, 'usr = ' > annc('Plugin, D)
             => < P : 'Plugin2 | 'b1 ; R , 'door = D, 'usr = ' > if D == 'L3 .


  rl [b2] : < P : 'Plugin2 | 'b1 ; R , 'door = D , 'usr = ' > pass'('Plugin, U, D)
             => < P : 'Plugin2 | 'b2 ; R , 'door = D, 'usr = U > .


  rl [b3] : < P : 'Plugin2 | 'b2 ; R , 'door = D > open('Plugin, D)
             => < P : 'Plugin2 | 'b3 ; R , 'door = D > .

  rl [b4] : < P : 'Plugin2 | 'b3 ; R , 'door = D > close('Plugin, D)
             => < P : 'Plugin2 | 'b4 ; R , 'door = D > .

  rl [b0] : < P : 'Plugin2 | 'b4 ; 'door = D, 'gsys = G , 'usr = U , R >
             => < P : 'Plugin2 | 'idle ; 'door = D , 'gsys = G , 'usr = ' > enter(G, D, U) .


--------------
```

```
  crl [c1] : < P : ’Plugin2 | ’idle ; ’door = D , R > pushin’(’Plugin, D)
            => < P : ’Plugin2 | ’c1 ; ’door = D , R > if D == ’L3 .

  rl [c2] : < P : ’Plugin2 | ’c1 ; R , ’door = D > open(’Plugin, D)
            => < P : ’Plugin2 | ’c2 ; R , ’door = D >  .

  rl [c3] : < P : ’Plugin2 | ’c2 ; R , ’door = D > close(’Plugin, D)
            => < P : ’Plugin2 | ’c3 ; R , ’door = D > .

  rl [c0] : < P : ’Plugin2 | ’c3 ; ’gsys = G , ’door = D, R >
            => < P : ’Plugin2 | ’idle ; ’gsys = G , ’door = D , R > exit(G, D) .

endm


mod GUIDESYSTEM1 is
  extending SOUP .
  protecting MID-LIST .

  op makeGSys : MachineId -> Machine [ctor] .
  var G : MachineId .
  --- Because there is only one gsys in this system, the MId is set to constatn ’Gsys
  eq makeGSys(G) = < G : ’GuideSys | ’0_N ; ’queue = nil , ’isbusy = 0 , ’qlen = 0 > .

  op tog : Bool -> Event .

  --- states: ’0_N, ’1_N, ’0_Y, ’Q_Y, ’Q_N
  var R : Attributes .
  vars U S D C : MachineId .
  var Q : MIdList .
  var L : Int .
 ---- the design changed..

  --- here we need to resolve the machine ids. What will happen if htere is more than one
  --- safe. the gsys need to keep one queue for each room. annc P for 1 plugin but D are
  --- different Doors


  rl [arr1] : < G : ’GuideSys | ’0_N ; ’queue = nil , ’qlen = 0, R > arrive(G, D, U)
            => < G : ’GuideSys | ’Q_N ; ’queue = (U) , ’qlen = 1 , R > .

  rl [arr2] : < G : ’GuideSys | ’0_Y ; ’queue = Q , ’qlen = L , R >  arrive(G, D, U)
            => < G : ’GuideSys | ’Q_Y ; ’queue = (Q U) , ’qlen = (L + 1), R >  .

  rl [arr3] : < G : ’GuideSys | ’Q_N ; ’queue = Q , ’qlen = L , R >  arrive(G, D, U)
            => < G : ’GuideSys | ’Q_N ; ’queue = (Q U), ’qlen = (L + 1), R >  .

  rl [arr4] : < G : ’GuideSys | ’Q_Y ; ’queue = Q , ’qlen = L , R >  arrive(G, D, U)
            => < G : ’GuideSys | ’Q_Y ; ’queue = (Q U), ’qlen = (L + 1), R >  .


  rl [annc1] : < G : ’GuideSys | ’Q_N ; ’queue = Q ,  R > tog(false)
            => < G : ’GuideSys | ’Q_N_temp ; ’queue = Q, R > annc(’Plugin , ’L3 )
            announce(head(Q), ’L3) topqueue(’L3, head(Q)) tog(true)  .

------------ prelude.maude is modified in line1007 for head(List{X})

  rl [exit1] : < G : ’GuideSys | ’Q_Y ; ’isbusy = 1, R > exit(G,D)
            => < G : ’GuideSys | ’Q_N ; ’isbusy = 0, R > .


  rl [exit2] : exit(G, D) < G : ’GuideSys | ’0_Y ; ’qlen = 0, ’queue = nil , ’isbusy = 1 >
            => < G : ’GuideSys | ’0_N ;  ’qlen = 0, ’queue = nil, ’isbusy = 0 > .


  rl [ent1] : < G : ’GuideSys | ’Q_N_temp ; ’queue = Q , ’qlen = 1, ’isbusy = 0 >
              enter(G,D,U) tog(true)
            => < G : ’GuideSys | ’0_Y ; ’queue = nil , ’qlen = 0, ’isbusy = 1 >
                tog(false) .
  crl [ent2] : < G : 'GuideSys | 'Q_N_temp ; 'queue = Q , 'qlen = L , 'isbusy = 0 >
                enter(G,D,U) tog(true)
              =>
                < G : 'GuideSys | 'Q_Y ; 'queue = tail(Q) , 'qlen = (L + (- 1)) , 'isbusy = 1 >
                tog(false)
if L > 1 .

endm



--- Policy
mod POLICY1 is
  extending SOUP .
  protecting ROOM1 .
  op makePolicy : MachineId -> Machine [ ctor ] .

  var M : MachineId . var R : Attributes .
  eq makePolicy(M) = < M : 'Policy | 's0 ; null > .

  var P : MachineId . vars X X' : Loc . var E : Event . var S : StateId .
  var W : Memory .

  rl ^( step(M, S, R, X, X') ) < P : 'Policy | 's0 ; null > => < P : 'Policy | 's0 ; null > step(M, S, R, X, X')  .
endm




---- User

fmod ULPAIR is
  protecting QID .
  sort ULPair .
  op _@_ : Qid Qid  -> ULPair [ctor prec 29] .
  op null : -> ULPair [ctor] .
endfm

fmod HISTORY is
  sort History .
  protecting ULPAIR .
  subsort ULPair < History .
  op H* : -> History [ ctor ] .
  op _&_  : History History -> History [ ctor assoc id: null prec 30 ] .

endfm



mod USER1 is
  extending SOUP .
  protecting ROOM1 .
  protecting HISTORY .
  op makeUser : MachineId -> User [ ctor ] .

  op takeoff : MachineId Loc -> Event .
  op look : MachineId Loc  -> Event .
  op mov : MachineId Loc Loc -> Event .
  vars M U : MachineId . var R : Attributes .
  eq makeUser(M) = < M : 'User | L('L0) ; 'atPosition ; null > .

  op updateM : Memory Loc Loc -> Event .

  ops actcounter actcounter1 : MachineId -> Event .

  vars X X' : Qid . var XL : List{Loc} . var W : Memory .
  var H : History .

  eq updateM(W, L(X), L(X')) = move(W, L(X), L(X')) . --- op move in module MEMORY



--- calculat the next position
```

```
  var S : StateId .

  rl [step] : step(M, S, R, L(X),L(X’)) < M : ’User | L(X) ; ’temp ; R >
      => < M : ’User | L(X) ; ’query ; R > look(M, L(X’))  .


--- from initial state \AtPosition\ there are
  crl [qr] : < M : ’User | L(X) ; ’atPosition ; R >
     => ^ step(M,’atPosition, R, L(X), choose( next(L(X)) ) ) < M : ’User | L(X) ; ’temp ; R >
        if  X == ’L0 or X == ’L4 .


--- query the intended position
  crl [qry] : look(M, L(X’)) < M : ’User | L(X) ; ’query ; R > W
              => < M : ’User | L(X) ; ’ready ; R > W takeoff(M,L(X’)) if space?(W,L(X’)) == true .

  crl [qrn] : look(M,L(X’)) < M : ’User | L(X) ; ’query ; R > W
              => < M : ’User | L(X) ; ’query ; R >  look(M,L(X’)) W if space?(W,L(X’)) == false .

--- move to intended position
  rl [mv2post] : takeoff(M,L(X’)) < M : ’User | L(X) ; ’ready ;  R > W H
            =>  < M : ’User | L(X’) ; ’atPosition ; R > H & M @ X’
    actcounter(M) updateM(W, L(X), L(X’))  .

  ---rl [staypost] : takeoff(M, L(X’)) < M : ’User | L(X) ; ’ready ;  R >
  ---          => ^ step(M, ’ready, R, L(X), choose(next(L(X))))
  ---            < M : ’User | L(X) ; ’temp ;  R > .



--- some user actions at the position
  crl [swipecard] : < M : ’User | L(X) ; ’atPosition ;  R > actcounter(M)
            => < M : ’User | L(X) ; ’atPosition ; R > swipe(X, M)
    if X == ’L1 or X == ’L3 .

  crl [swipecard] : < M : ’User | L(X) ; ’atPosition ;  R > actcounter(M)
            => < M : ’User | L(X) ; ’atPosition ; R >
    if X =/= ’L1 or X =/= ’L3 .

  crl [rdrprv] : < M : ’User | L(X) ; ’atPosition ;  R > readerAck(M, true)
            => < M : ’User | L(X) ; ’atPosition ; R > actcounter1(M)
    if X == ’L1 or X == ’L3  .

  crl [push] : < M : ’User | L(X) ; ’atPosition ;  R > actcounter1(M)
            => < M : ’User | L(X) ; ’inAction ; R > push(X, M)
    if X == ’L1 or X == ’L3  .

 --- the user is already removed from the waiting listin the Guide System,
 --- if he did not pass then send him home
 --- crl [rdrnprv] : < M : ’User | L(X) ; ’atPosition ;  R > W H
 ---         readerAck(M, false) => ^ step(M, ’atPosition, R, L(’L6), choose(next(L(’L6))))
 ---         < M : ’User | L(X) ; ’temp ;  R >
 ---         updateM(W, L(X), L(’L6)) H & M @ X’ .


  crl [pushfrin] : < M : ’User | L(X) ; ’atPosition ;  R >
            => < M : ’User | L(X) ; ’inAction ; R > pushin(D(X), M)
    if X == ’L5 or X == ’L6  .

---  rl [heisastone] : < M : ’User | L(X) ; ’atPosition ;  R >
---            => ^ step(M,’atPosition, R, L(X), choose( next(L(X)) ) )
---              < M : ’User | L(X) ; ’temp ;  R > .



  crl [drpr] : < M : ’User | L(X) ; ’inAction ;  R > doorAck(M, true)
            => ^ step(M,’inAction, R, L(X), choose( next(L(X)) ) )
      < M : ’User | L(X) ; ’temp ;  R >
    if X == ’L1 or X == ’L3 or X == ’L5 or X == ’L6 .
  crl [drnpr] : < M : ’User | L(X) ; ’inAction ;  R > doorAck(M, false)
            => < M : ’User | L(X) ; ’atPosition ;  R >
    if X == ’L1 or X == ’L3 or X == ’L5 or X == ’L6 .



  crl [annc] : < M : ’User | L(X) ; ’atPosition ;  R > announce(M, X’)
            => ˆ step(M,’atPosition, R, L(X), L(X’) )
       < M : ’User | L(X) ; ’temp ;  R >
    if X == ’L2 and X’ == ’L3 .


---  crl [hehasawalk] : < M : ’User | L(X) ; ’atPosition ;  R >
---            => ˆ step(M,’atPosition, R, L(X), choose( next(L(X)) ) )
---            < M : ’User | L(X) ; ’temp ;  R >
---    if X == ’L0 or X == ’L4 .

endm


--- setup environment and test

cd ../../Documents\ and\ Settings/root/My\ Documents/My\ Project/secsys

in LocGraph.maude
in StateMachine.maude
in UserPolicy.maude
in PolicyEnforcer.maude
in secsys.maude
--- set trace on .

mod test is
  protecting DOOR1 .
  protecting CARDREADERW1 .
  protecting CARDREADERW2 .
  protecting PLUGIN1 .
  protecting PLUGIN2 .
  protecting PLUGIN3 .
  protecting GUIDESYSTEM1 .
  protecting USER1 .
  protecting POLICY1 .

  protecting MEMORY .

  ops test0 test1 test2 test3 test4 test5 test6 test7 test8 test9 : -> Soup .
  ops test10 test11 test12 test13 test14 test15 test16 test17 test18 test19 : -> Soup .
  ops test20 test21 test22 test23 test24  test25 test26 test27 test28 test29 : -> Soup .


  eq test0 =  m(inc(inc(nobody, L(’L0)), L(’L5))) .

  eq test1 =  m(inc(inc(nobody, L(’L0)), L(’L2))) .

  eq test2 = makeDoor(’Waiting) pushin(’Waiting, ’me) .

  eq test3 = test2 test1 .

  eq test4 = swipe(’Waiting, ’me) makeCardReaderW1(’Waiting, (’34 ’3 ’me)) .

  eq test5 = swipe(’Waiting, ’me) makeCardReaderW1(’Waiting, nil) makeDoor(’Waiting) .

  eq test6 = topqueue(’Work1, ’me) swipe(’Work1, ’me) makeCardReaderW2(’Work1) .

  eq test7 = makePlugin2(’Plugin1, ’L3, ’Gsys ) pushin’(’Plugin, ’L3) open(’Plugin, ’L3) close(’Plugin, ’L3) .

  eq test8 = makePlugin1(’Plugin2, ’L3,  ’Gsys) pass’(’Plugin, ’u0, ’L1) close(’Plugin, ’L1) open(’Plugin, ’L1) .

  eq test9 = makePlugin2(’Plugin3, ’L3,  ’Gsys) annc(’Plugin, ’L3) pass’(’Plugin, ’u0, ’L3) open(’Plugin, ’L3) close(’


  eq test10 = makeGSys(’d) arrive(’d, ’L1, ’U1) arrive(’d, ’L1,’U2) arrive(’d, ’L1,’U3) tog(false) .

  eq test11 = makeGSys(’d) arrive(’d, ’L1, ’U1) arrive(’d,’L1, ’U2) arrive(’d, ’L1, ’U3) tog(false) enter(’d, ’L3, ’U1
  eq test12 = makeGSys('d) arrive('d, 'L1, 'U1) arrive('d, 'L1,'U2) tog(false) enter('d, 'L3, 'U1) .

  eq test13 = makeGSys('d) arrive('d, 'L1, 'U1) arrive('d, 'L1,'U2) tog(false) enter('d, 'L3, 'U1) exit('d, 'L3) .

  eq test14 = makeGSys('d) arrive('d, 'L1, 'U1) tog(false) enter('d, 'L3, 'U1) exit('d, 'L3) .



  eq test15 = makeUser('u1) m(inc(nobody, L('L0))) .

  eq test16 = makeUser('u1) m(inc(nobody, L('L0))) makePolicy('p) H* .

  eq test17 = makeUser('u1) m(inc(nobody, L('L0))) readerAck('u1, true) makePolicy('p)  H*  .

  eq test19 = makeUser('u1) makeUser('u2) m(inc(inc(nobody, L('L0)), L('L0)))
            readerAck('u1, true) readerAck('u2, true) doorAck('u1, true)
      makePolicy('p)  H*  .


  eq test25 = makeUser('u1) m(inc(nobody, L('L0)))
            makeDoor('L1) makeCardReaderW1('L1, ('u1 'u2 'u3))
      makeDoor('L3) makeCardReaderW2('L3)
      makePlugin1('Plugin1, 'Gsys, 'L1)
      makePlugin2('Plugin2, 'Gsys, 'L3)
      makeGSys('Gsys) tog(false)
      makePolicy('p)  H*  .
      ---make sure the plugin and the gsystem has the same id

  eq test26 = makeUser('u2) makeUser('u1)  m(inc(inc(nobody, L('L0)), L('L0)))
            makeDoor('L1) makeCardReaderW1('L1, ('u1 'u2 'u3))
      makeDoor('L3) makeCardReaderW2('L3)
      makePlugin1('Plugin1, 'Gsys, 'L1)
      makePlugin2('Plugin2, 'Gsys, 'L3)
      makeGSys('Gsys) tog(false)
      makePolicy('p)  H* .
      ---make sure the plugin and the gsystem has the same id

  eq test27 = makeUser('u2) makeUser('u1)
makeUser('u3) makeUser('u7)
makeUser('u4) makeUser('u6)
makeUser('u5)
            m(inc(inc(inc(inc(inc(inc(inc(nobody,
L('L0)), L('L0)), L('L0)), L('L0)), L('L0)), L('L0)), L('L0)))
      makeDoor('L1) makeCardReaderW1('L1, ('u1 'u2 'u3 'u4 'u5 'u6 'u7))
      makeDoor('L3) makeCardReaderW2('L3)
      makePlugin1('Plugin1, 'Gsys, 'L1)
      makePlugin2('Plugin2, 'Gsys, 'L3)
      makeGSys('Gsys) tog(false)
      makePolicy('p)  H* .
      ---make sure the plugin and the gsystem has the same id

endm



‘
```