Formal Verification of the MUD case study

Stefania Gnesi and Franco Mazzanti

Istituto di Scienze e Tecnologie dell'Informazione "A.Faedo", C.N.R. – Pisa (Italy)

Abstract: In this paper we present the latest advances of the UMC verification environment and the results of its experimental application to the AGILE MUD case study.

1. An overview of UMC

The early experience gained with the UMC V.2.5 [1] prototype and the airport case study [2] [3] has driven the evolution of the tool in several directions:

The *model definition language* has been extended and enriched, allowing to explicitly define a system deployment phase (object instantiations) in addition to the statechart definitions phase (i.e. statechart are now associated to classes, and not direclty to objects). Moreover, the support of some missing important UML features, like synchronous call operations (and function calls), and deferred events has been introduced.

The *temporal logic supported* by the tool has been partly redesigned, switching the enphasis from mu-ACTL [4] to standard mu-calculus enriched with higher level (weak) temporal opeartors (in CTL/ACTL-like style), and with user definable state predicates; the new logic being called mu-UCTL [5].

The web tool interface has been redesigned, trying to increase the tool usability, and several new model exploration and analysis features have been introduced. Among these the possibility to render the possible model evolutions in the form of an SVG chart, the possibility to select the desired level of abstraction while visualizing the system evolutions , the possibility to generate abstract minimized views of the system reflecting specific aspects of interest of the model, and possibility to visualize the minimal-depth fragment of the model with satisfies or falsifies a given temporal logic formula (example or counter-example).

The current version of UMC (now V3.1) is still a running prototype, useful for investigating and experiencing possible improvents of the state of art, and definitely usable for didactic purposes, while performce aspects and capability of handling extremely large systems are two aspects to which has not yet been devoted the needed attention as a widely usable tool would need. The current version of UMC can be tried online at the url *http://matrix.isti.cnr.it/umc/V3.1/umc.html*, and the discussed MUD case study can be browsed by selecting “simplemud” from the predefined examples list (Model Definition .. --> View a UMC Sample Model …).

As a remainder, we summarise here the various assumptions and limitations under which UMC currently works).

--------------------- at system level --------------------

A system is constituted by fixed static set of objects (no dynamic active object creation).
A possible object evolution becomes a possibile system evolution (i.e. the parallel composition of objects is modelled through interleaving).
The communications (exchange of signals and call operations) between objects of the system occur without delays, loss, or re-orderings. Communication is direct and one-to-one (no broadcasts).
The events queues associated to objects are simple FIFO queues.

---------- at the level of class statecharst -----------

The only supported types for the class variables and event parameters are boolean, integer and object.
The only actions which can appear in state transitions are assignments (to local variables), sending of signals, synchronous (function) call operations (and return stmts).
History / Deep History, and Synch states are not supported.
Dynamic choice / Static choice composite transitions are not supported
Change events, Time events are not supported.
Satetchart refinements, statechart inheritance and subcharts are not supported.

------------ are instead currently supported -------------

Composite and Parallel states
Fork and Join transitions
Synchronous and Asynchronous calls
Local variables and assignments
Completion events and completion transitions
Deferred events
The standard UML transition priority mechanism.
The standard UML "run to completion step" semantics

2. An overview of MUD case study

The MUD case study is the proposal to specify and verify a system behaving like a role playing game. We have players which have certain attributes (e.g. strength), and which move across the rooms of a maze trying to reach a given place., and we have rooms (which may contain objects or “monsters”, or be empty). The player in a room can interact with it accordingly to their respective state. A player can dye, during the fight with a monster, which implies restarting the game from the beginning after leaving all the collected objects in the room where he has been defeated, or can complete the game by reaching the required place.

Several versions of this kind of system have been produced, and the version we are presenting here is a carefully simplified model. We believe in fact that easyness of understanding is indeed a very important factor which should guide the model definition as far as possible provided that it allows to experiment the analisys of all the intended kinds of properties. The full code of the model classes (as taken by UMC) is shown in Appendix A. The examples of system instantiations are shown in Appendix B, the set of formulas related to this kind of system are listed in Appendix C.

3. The details of the simplyfied system

We have two classes, corresponding to the two kinds of active objects in the system: class Room and class Player. A System is supposed to be constituted by a set of Room instantiations plus a set of Player instantiations.

A Room object is characterized by a list of other rooms to which it is connected (at most three), by the presence (or not) of a monster with given stength and healtness, by the presence (or not) of a player inside it, by the presence (or not) of an object inside it (i.e. a sword). At each step, if a moster and a player are present in the room, the room activity starts with the communication to the player of the room content and of the damage inflicted to him by the monster. If the player is engaged in a fight and survives from the attack he must respond to the fight with another attack (until one of the moster or the player is defeated). If the room contains only the player (no monster originally present or the monster having been defeated) then the player can perform any other action among exiting from the room and entering in the next, taking some object, or completing the game. If a room is empty it is inactive, waiting for a player to enter.

A Player object is characterized by the room from which the game is started, from the room where the game becames successfully completed, by the current location (the current room), the initial and current healtness, and the current strenght (capacity of inflicting damage). Once entered in a room its activity consists in performing a cycle of observing the room content and doing an activity until the player dies or exits from the room for entering into another room.

The statecharts for class Room and Player are shown in Figure 1 and 2.

The class diagram for classes Room and Player are shown in Figure 3

The object diagrams for a simple system constituted by one player and four room is shown in Figure 4. This system is the one being used for the analysis in the following Sections.

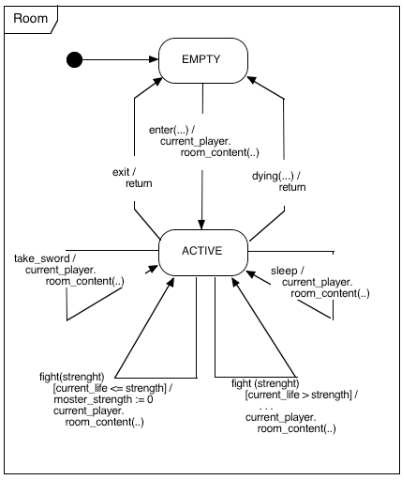
Room
EMPTY
enter(...) / current_player. room_content(..)
exit / return
dying(...) / return
ACTIVE
take_sword / current_player. room_content(..)
sleep / current_player. room_content(..)
fight(strenght) [current_life <= strength] / moster_strength := 0 current_player. room_content(..)
fight (strenght) [current_life > strength] / . . . current_player. room_content(..)

Figure 1. Statechart of Room

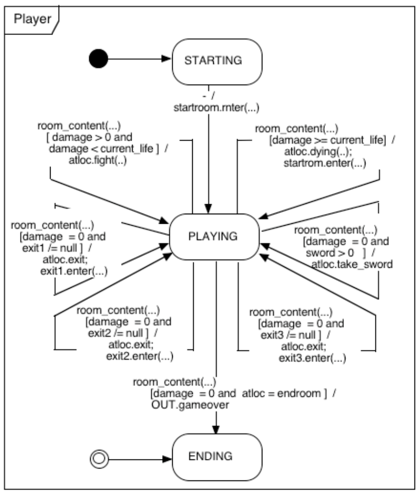
Player
STARTING
- / startroom.rnter(...)
room_content(...) [damage > 0 and damage < current_life] / atloc.fight(..)
room_content(...) [damage >= current_life] / atloc.dying(..); startrom.enter(...)
PLAYING
room_content(...) [damage = 0 and exit1 /= null] / atloc.exit; exit1.enter(...)
room_content(...) [damage = 0 and sword > 0] / atloc.take_sword
room_content(...) [damage = 0 and exit2 /= null] / atloc.exit; exit2.enter(...)
room_content(...) [damage = 0 and exit3 /= null] / atloc.exit; exit3.enter(...)
room_content(...) [damage = 0 and atloc = endroom] / OUT.gameover
ENDING

Figure 2. Statechart of Player

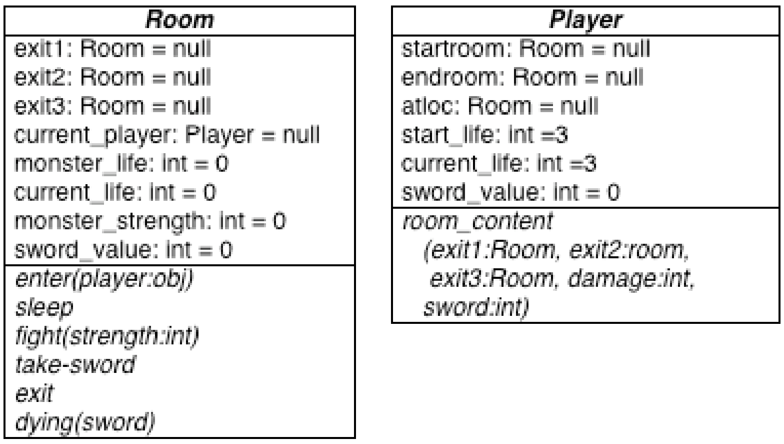

Figure 3. Class diagram for Room and Player

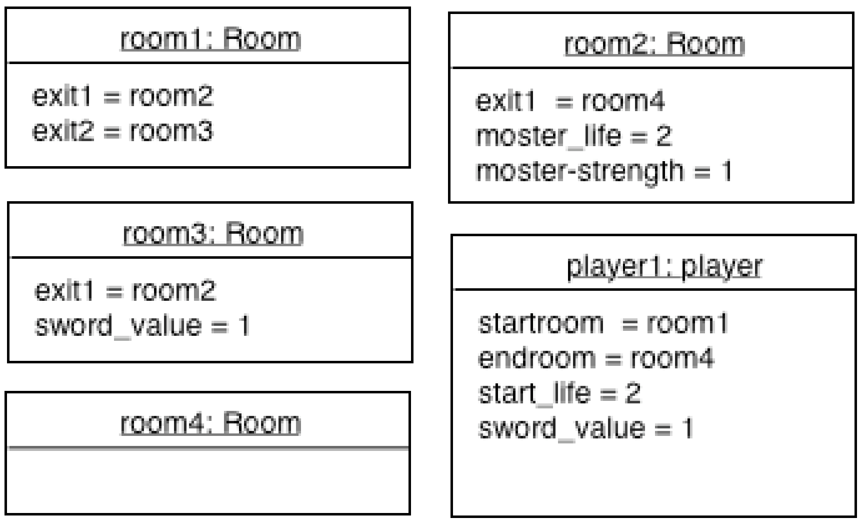

Figure 4. Object diagram for system with 4 rooms and 1 player

4. Exploring the model

When a new specification is being built, two distinct verification phases are likely to occur. A first phase, somewhat similar to the debugging phase of software development, is that one in which the user tries to get confident that in the design there are no macroscopic erros and the overall specification actually matches what more or less he/she had in mind. This first debugging / overviewing phases are supported by UMC with the possibility of browsing the system evolutions asadoubly labelled transition system and by the possibility of viewing "minimized model abstraction", i.e. a version of the doubly labelled transition system (modelling all the system evolutions) in which only some specified aspects are kept visible, while all the non-essential aspects are hidden (i.e. info is removed from state and edegs labels), and where the abstracted graph is then minimized according to complete-trace, divergence-sensitive equivalence.

In other words we just want to be able to observe all the possibile full system traces abstracting from all the intermediate steps which are non relevant w.r.t. what we want to observe.

For example, if we take a “black-box” like observation mode for the system shown in Figure 4 (i.e. we only want to observe the signal which exit to the outside of the system) we can build a model abstraction in which we do not observe any internal detail, and the resulting model abstraction becomes that one shown in Figure 5.

In this case we have the immediate feedback that the overall system behavior consist either in looping performing some internal activity, or in successfully terminating (e.g. there are no deadlocks).

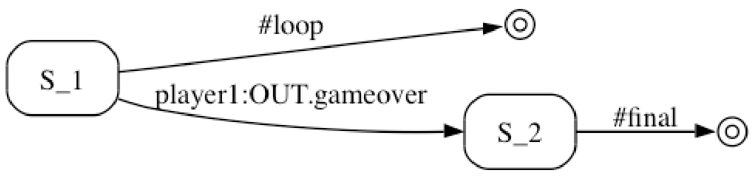

Figure 5. A black box abstraction of the system

Suppose instead that we just want to observe to the “sword_value” attribute of object “room2”. We can customize the UMC abstraction interface specifying that we are interested in observing that specific object attribute, and build the corresponding model abstraction. The result is shown in Figure 6.

We already know that room2 does not initially contain any sword (hence its value is initially 0), but from the chart we can also see that, in order to complete the game that initial value must not change. As soon as the sword_value becomes 1 we enter into a subchart of the system evolution from which it is no longer possible to win. Notice that in order for sword_value to become “1” we need that a player (with sword_strength=1) enters room2 and lose the fight with the local moster, once the player dies in room2, his sword is discarded there and the player will nevermore be able to defeat the moster.

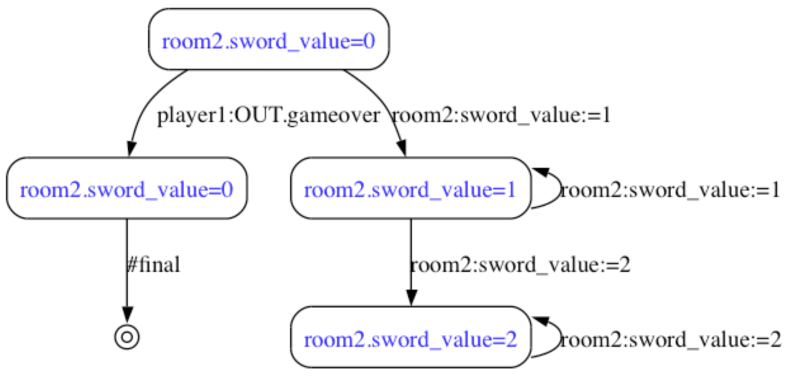

Figure 6.: observing the evolutions of "sword_value" of "room2".

Suppose finally, that we want to observe all the possible moves of player1 across the rooms. We can specify the visibility of the "atloc" atrribute of player1 and build the corresponding model abstraction. We will obtain the picture of Figure 7.

From that chart we can see that the only chance for player1 to win is to move from room1 to room3 (and there to take an additional sword), from room3 to move to room2 (and defeate the moster) finally reaching room4.

The interesting part of this approach is that the user does not need any specific logical / technical ability for building useful abstractions of the system and validate the system behavior. Another interesting aspects of this approach is that in this way it is easier to "discover the unexpected". In other words we might even not have a precise idea of which properties should our model satisfy, and still obtain some reasonably simple scenarious to analyse which can point out strange, wrong, or simply unexpected aspects of the system behavior.

5. Proving the properties

A problem of the previous approach is that, if the system diverges (i.e. in has an infinite number of states) we are no longer able to compute the full-trace minimized model. This kind of operation, in fact requires complete global view of the system.

Even without diverging, the system may be simply too big for allowing the abstract minimized model to be built in a reasonable time using the available resources.

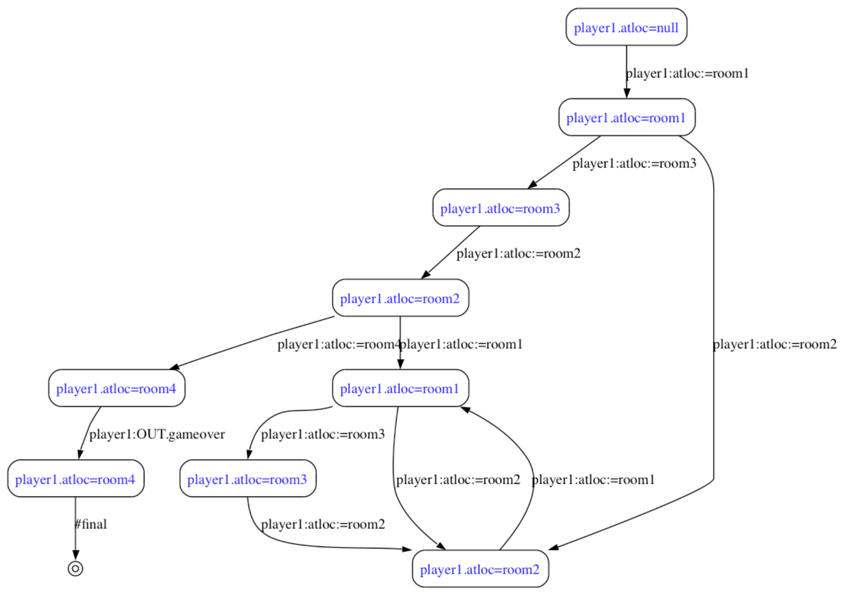

Figure 7.. Observing “atloc” evolutions of the system

If we succed in building the minimized model, and the resulting chart is too complex (e.g. it may already be sufficient the fact that the graph does not fit into a single page) the possibility of being able to “validate” it by just looking at it might be a naive illusion.

Finally, even if the abstract model exist, and is reasonably simple, we might be interested in an official formal answer (e.g. to be associated to the design as documental evidence) stating that a certain property is satisfied, rather then a “resonably simple” graph from which “some expert eyes were convinced” that a certain property holds.

Last, but not least, in the case we identify some wrong or unexpected behavour in the specification, we would definitely be interested in observing the details of the system model at the specific point in which the anomaly is identified. Unfortunately, while reasoning on the abstract minimized view we are no longer able to map the abstract state to the original system configuration. In order to do that we need to reason on the abstract, but not minimized system as produced by the UMC exploration feature, or, as a better solution, to the minimal explanation submodel generated by UMC after the evaluation of a formula (example or counter-example).

All this to say that the possibility of building abtractions of the system is surely a very useful and important feature, but that this possibility is also likely to be just the first step of a more detailed system verification phase.

The next phase in fact is that one in which specific or more advanced properties of the model are checked, and feedback from the evaluation analised.

For example, with respect to the system instantiation shown in Figure 4, we might want to verify that “s it actually possible for player1 to complete the game.
This can be done by checking the formula:

-- possibility of game completion (True)
```
EF <gameover> true
```

Moreover, we might want to verify that there is no way to reach a final state other than successfully completing the game (i.e. there are no deadlocks).
This can be checked with the formula:

-- It is possible to deadlock (False)
```
E[ true {~gameover} U FINAL]
```

Another property which we might want to verify, as suggested by the previous analysis, is that it becomes impossible to complete the game if the player ever loses a fight (hence, if the player dies).

-- It is possible to win after having being killed (False)
```
E [ true { dying } U FINAL ]
```

Another property realated to the maze traversal might be the following.

-- It is possible to complete the game without
-- ever passing from Room3 (False)
```
E [ ~ ASSERT(player1.atloc=room3) {true} U FINAL ]
```

While a property related to object manipulation might be the following:

-- It is possible to win without taking the sword (False)
```
E[ true {~take_sword} U FINAL]
```

Another property involving objects and locations might be

-- A new sword can only be taken while in room3 (True)
```
AG ((<take_sword>true) -> ASSERT(player1.atloc=room3))
```

A typical branching time formula (of which we already know the result because of the previous analisys) could be the following:

-- From every point of the game it is possible
-- to successfully complete it (False)
```
AG EF FINAL
```

Once the player succeedes in taking the sword without dying, from that point on it is always possibile to complete the game.

-- After taking the sword without dying we always can win (True)
```
   A[ true { ~ dying } U [ take_sword ] AF FINAL ]
```

The following is a formula which, inspite of its simplicity, requires the fix point operator for being expressed:

-- It is possible to dye infinitely many times (True)
```
   max Z: EF <dying> Z
```

Finally let us check a topologic invariant for the system: i.e. if player1 thinks to be in room1, then room1 agrees that player1 is inside itself.

-- there is always topologic agreement between room1 and player1
```
AG (
    (ASSERT(player1.atloc=room1) &
       ASSERT(room1.current_player=player1)
    )  |
   (
      ~ASSERT(player1.atloc=room1) &
       ~ASSERT(room1.current_player=player1)
    )
  )
```

We might at first be surprised to see that this property does not hold.
If we look at the explanation (model) for this formula (omitted here for its simplicity) we can see that already at configuration2 (after just one step of system evolution) the invariant is false. The explanation is simply that player1 sends a signal to room1 requesting to enter, but until that signal is dispatched by room1 there is a temporary "topological disagreement" between these two objects.

6. Conclusions and future works

The application of UMC to the MUD case study has provided very useful feeback for the evaluation of the development effort done so far and for the planning of future developments. Apart for continuously improving the tool usability, and the coverage of UML features (for example, parameterized classes and state activitites), there are some directions of work which also this case study seem to encorage.

Currently only a few simple and flat data types (int, obj, bool) are supported by UMC . Surely the possibility of using a few predefined structured data types (as sets or vectors) would greatly help the task of designing system spefications.

Since version 2.5 of UMC, the feature of model extraction from a standard XMI definition of an UML model has not been updated reflecting all the evolutions of the tool. Indeed this capability of interacting with other standard UML environment is important, and this feature should eventually be consistently restored.

As the “topological invariant” study as shown, it is probably useful to have to possibility to access and modify some kind of “global data space” (e.g. for holding a global topological map for the system) shared among all objects.

At the system level, it might be useful to allow the user to select different kinds of scenarios. e.g non just simple and pure interleaving among objects, but also other “more efficient” scheduling schemas (maybe taking into account fairness principles or priority mechanisms); non just immediate and safe communications but possibly also unsafe/ delayed inter-object communications.

At the logic level many improvement could be thought to be supported by UMC: starting from a better support of data values in transition actions and state predicates to the possibility of selecting different logical scenarious like CTL*/ACTL*, LTL, TLA o MTLA.

7. References

[1] Franco Mazzanti "UMG User Guide (Version 2.5)" ISTI Technical Report 2003-TR-22 , September 2003

[2] L. Andrade, P.Baldan, H.Baumeister, R.Bruni, A.Corradini, R.De Nicola, J.L.Fiadeiro, F.Gadducci, S.Gnesi, P.Hoffman, N.Koch, P.Kosiuczenko, A.Lapadula, D.Latella, A.Lopes, M.Loreti, M.Massink, F.Mazzanti, U.Montanari, C.Oliveira,R.Pugliese, A.Tarlecki, M.Wermelinger, M.Wirsing, and A.Zawlocki:"AGILE: Software Architectures for Mobility" in Recent Trends in Algebraic Development, LNCS 2755, Springer Verlag, November 2003

[3] Stefania Gnesi and Franco Mazzanti “On the fly model checking of communicating UML State Machines” in Proceedings of Second ACIS International Conference on Software Engineering Research, Management and Applications, May 5-7 2004, Los Angeles, USA.

[4] Stefania Gnesi and Franco Mazzanti “Mu-ACTL+: A temporal logic for UML Statechart diagrams”. ISTI 2003-TR-64 November 2003.

[5] S.Gnesi , F.Mazzanti, “Modal Logics for Behavioural Properties of Mobile Systems” Deliverable D2.3b project AGILE, 2003

Appendix A: UMC code of model classes

```
Class Room is
Signals:  enter(player:obj), sleep,
          fight(strength), take_sword
Operations: exit, dying(sword)
Vars: exit1:obj, exit2:obj, exit3:obj, current_player:obj,
      monster_life, current_life, monster_strength, sword_value

State Top = EMPTY, ACTIVE
Defers: enter(player:obj)      -- in case of multiple players

Transitions:
 --
EMPTY -( enter(player:obj) /
    current_player := player;
    current_life := monster_life;
    current_player.room_content(exit1, exit2, exit3,
                               monster_strength, sword_value)
    )->  ACTIVE
 --
ACTIVE -( exit /
    current_player := null;
    return )-> EMPTY
 --
ACTIVE -( dying(sword) /
    sword_value := sword_value + sword;
    current_player := null;
    return )-> EMPTY
 --
ACTIVE -( take_sword /
   sword_value :=0;
    current_player.room_content(exit1, exit2, exit3,
                               monster_strength, sword_value)
   )-> ACTIVE
 --
ACTIVE -( fight(strength) [current_life > strength ]/
    current_life := current_life - strength;
    current_player.room_content(exit1, exit2, exit3,
                               monster_strength, sword_value)
   )-> ACTIVE
 --
ACTIVE -( fight(strength) [current_life <= strength ]/
    current_life := 0;
    monster_strength :=0;
    current_player.room_content(exit1, exit2, exit3,
                            monster_strength, sword_value)
    )-> ACTIVE
 --
ACTIVE -( sleep /
    current_player.room_content(exit1, exit2, exit3,
                            monster_strength, sword_value)
    )-> ACTIVE
end Room
```

```
Class Player is
Signals:  room_content(exit1:obj, exit2:obj, exit3:obj,
                       damage, sword)
Vars: startroom:obj, endroom:obj, atloc:obj,
      start_life:=3, current_life:=3, sword_value:=0

State Top =  STARTING ,  PLAYING, ENDING

Transitions:
STARTING -( - /
    atloc := startroom;
    current_life := start_life;
    startroom.enter(self) )-> PLAYING
 --
PLAYING -( room_content(exit1,exit2,exit3, damage, sword)
 [damage >= current_life] /
    atloc.dying(sword_value);
    current_life := start_life;
    sword_value := 0;
    atloc := startroom;
    startroom.enter(self) )-> PLAYING
 --
PLAYING -( room_content(exit1,exit2,exit3, damage, sword)
 [(damage < current_life) and (damage > 0)] /
    atloc.fight(sword_value);
    current_life := current_life - damage )-> PLAYING
 --
PLAYING -( room_content(exit1,exit2,exit3, damage, sword)
 [(damage = 0) and (exit1 /= null)] /
    atloc.exit;
    atloc := exit1;
    exit1.enter(self) )-> PLAYING
 --
PLAYING -( room_content(exit1,exit2,exit3, damage, sword)
 [(damage = 0) & (exit2 /= null)] /
    atloc.exit;
    atloc := exit2;
    exit2.enter(self) )-> PLAYING
 --
PLAYING -( room_content(exit1,exit2,exit3, damage, sword)
 [(damage = 0) & (exit3 /= null)] /
    atloc.exit;
    atloc := exit3;
    exit3.enter(self) )-> PLAYING
 --
PLAYING -( room_content(exit1,exit2,exit3, damage, sword)
 [(damage = 0) & (sword > 0)] /
    atloc.take_sword;
    sword_value := sword_value + sword )-> PLAYING
 --
PLAYING -( room_content(exit1,exit2,exit3, damage, sword)
 [(damage = 0) & (atloc = endroom)] /
  OUT.gameover)-> ENDING
end Player
```

Appendix B: UMC code of system deployment (initial object instantiations)

```
-------------------- simplemud-4-1----------------------------
--
Object Room1: Room (Exit1 => Room2, Exit2 => Room3)
Object Room2: Room (Exit1 => Room4,
                    Monster_life => 2, Monster_strength => 1)
Object Room3: Room (Exit1 => Room2, Sword_value=>1)
Object Room4: Room
Object Player1: Player (Startroom => Room1, Endroom => Room4,
                        Start_life => 2,Sword_value=>1)
--
-------------------- end simplemud-4-1-------------------------
```