Architectural Modeling of Distributed Interactive Systems: The MUD Game Case-study

Nasreddine Aoumeur* Cristóvão Oliveira† José Luiz Fiadeiro

Department of Computer Science
University of Leicester
LE1 7RH, UK
{na80,co49,jwf4}@le.ac.uk

Abstract

Coordination and distribution of computations and interactions are still raising major challenges in the development of complex software-intensive systems. In this paper we use a case study to show how a component-based approach that promotes coordination and distribution as separate architectural concerns can lead to new levels of adaptivity, agility and openess in system construction and evolution. The case study is a variant of the so-called Multi-User Dungeon (MUD) game. We Illustrate how two semantic primitives – coordination and location laws – can be used to modelcoordination and distribution as separate concerns as they arise in the rules of game, and how they can be composed and evolved, at runtime, to reflect mobility and other changes on the configuration of the domain that result from playing the game.

Keyword: Composition, Coordination, Distribution, Interaction, Location-awareness, Mobility, MUD game, Reconfiguration, Superposition.

1 Introduction and motivation

It is widely recognized that *coordination* and *distribution* of computations and interactions are core concerns for achieving the higher levels of adaptivity, agility and openness required of software-intensive systems operating in the new ages of global, pervasive and ubiquitous computing. In order to cope with the new levels of complexity raised by these concerns, the AGILE [1] consortium [ABB+03]. [ABB+03] investigated the use of methods and techniques similar to those developed in the field of *Software Architecture* (e.g. Allen02,Garlan04,Nenad03). More precisely, we adhered to the use of architectural *connectors* [AG97] as first-class entities that coordinate interactions independently of the way components perform their local computations.

This separation between computation and coordination concerns has been extensively explored as a means of increasing the level of agility through which systems can evolve in reaction to changes in the business rules according to which they are required to operate

*Supported by the European Commission through the contract IST-2001-32747 (AGILE: Architectures for Mobility)

†Supported by FCT, Portugal, through the PhD Scholarship SFRH/BD/6241/2001.

[1]IST-FET-GC1 project, URL:http://www.pst.informatik.uni-muenchen.de/projekte/agile.

[AF02, AF03]. So-called "Coordination Laws" have been developed as semantic primitives for modelling business rules and other volatile aspects of the application domain. Their instances, called "Coordination Contracts", can be superposed dynamically over the system configuration without interfering with the computations that implement the core services that components provide. A software development environment supports the approach [AGKF02]. Its use in industrial-size projects has been documented [WKA+03]. The formal semantics of the approach has been developed over the architectural description language CommUnity using category theory [FLW03, Fia04].

The main challenge addressed in AGILE concerned the extension of this architectural approach to the *distribution* dimension. This concerns the way business rules or other aspects of the application domain depend on the *locations* where computations are performed and the properties of the *network* across which interactions need to be coordinated. For instance, when modelling a banking application, a simple operation like a withdrawal involves computational aspects that are quite stable (the way the account is debited), coordination aspects that can vary (e.g. the package that the customer has negotiated with the bank, including overdrafts and other business aspects that interfere with the withdrawal), as well as distribution aspects that are location-dependent (e.g. the business channel that is being used for the operation – at the counter, at an ATM, through the internet, and so on). The way the system behaves at any given state emerges from the composition of these three dimensions.

This is why we developed new conceptual modelling primitives for location concerns, what we have called *location laws*. Their instances are a new kind of architectural connectors called *location contracts* that superpose the dependencies of computations and interactions on distribution. These have been motivated and characterised at an intuitive level in [AFO04b]. More specifically, we demonstrated how a coordination/location-driven architectural approach can be put to an effective use in modelling distributed business processes within the service-oriented computing paradigm [AFO04a]. The semantics of this new architectural dimension has been defined over an extension of CommUnity [LFW02, FL04]. Meseguer's Rewriting Logic [Mes92] has also been used to capture the operational aspects and validation of the superposition of the three dimensions: computation, coordination and distribution [AF05].

The purpose of this paper is twofold. One the one hand, to present a comprehensive account of the approach around a single example, bringing together aspects that are scattered in shorter publications and illustrated with several different examples [AFO04b, AFO04a, FL04, **?**]. On the other hand, to show how the approach copes with the challenges raised by a totally different case study, one in which distribution and mobility are intrinsic to the "rules of the game": a Multi-User Dungeon (MUD).

The rest of this paper is organised as follows. In Section 2, we start by giving an informal description of the case-study. Then, we provide more motivation for the need to separate Coordination and Distribution as architectural dimensions. In Section 3, we present the semantic primitives that we developed for coordination and location aspects around a simple action of the MUD game. Sections 4, 5 and 6 detail how coordination and distribution concerns are separately modelled, and how they are integrated to account for each action of the game. The paper concludes by summarising what we consider to have been achieved and highlighting different directions for further work.

2 Overview of the MUD game

We present SWEPMUD as a multi-user dungeon (MUD) game that is played with mobile phones [BBH+04]. This is one of the case studies used in AGILE [ABB+03]. In this version, the player registers once for the game and can then use his/her cell phone to play. The "field of play" is partitioned into different levels. Each level consists of several rooms. The rooms are connected by doors that allow players to move from one room to another. Inside a room, a player can interact with other players as well as take, use, or release objects. The goal is to traverse all levels and complete with success the task of the special room at the final level.

The main entities involved in the game are the following:

Rooms Rooms are ranked in levels. For a given level, rooms can be organised in an undirected connected graph where the edges correspond to doors. There are two special rooms at each level: the starting room, which is also where players are thrown into when they "die", and a special room that gives access to the starting room of the next level.

Players Players have a number of characteristics such as strength, agility and life points. Inside a room, players can communicate with each other, trade objects for other objects or money, and fight each other using the weapons that they have available. Players can also move between rooms. When they die, they are moved to the starting room. When reaching the special room of one level, a player can move to the starting room of the next level.

Objects Objects reside in rooms and, depending on the circumstances, can be carried by players when they move between rooms. Objects are used by players for different purposes. For instance, healing potions give life points. Weapons, like swords, knives, shields and armours, can be used in a fight. Objects have states indicating their condition, which can change as players use them.

An architectural analysis of the different functional aspects of the game can be made in each of the three dimensions already motivated:

Computation By "computation" we mean any transformation that a component operates on its state such as players' life points and objects' condition. The separation of concerns that we are enforcing means that these operations cannot engage the component in interactions; there is no explicit invocation of services provided by other components; components can only use operations on data as made available by local libraries.

Coordination Interactions in this system concern the relationships player-to-player (e.g. while fighting or trading) and player-to-object (e.g a player using a weapon). The way these relationships need to be coordinated derives from the rules of the game and is captured in "laws" that can be superposed at runtime according to the "state of play". Interactions are triggered by events published by components or the system environment.

Distribution Rooms are clear candidates for playing the role of locations because the way components behave and interact depends on the room in which they are located. For instance, movement (of players) is possible only between neighbouring rooms endowed

with doors. Trading, fighting, and talking (between players) are only possible within the same room.

We claim that these aspects should be modelled independently of one another at design time, and brought together at runtime as required by the "state of play". For instance, the transformations through which player's life points are updated depend only on the local representation of the player's data. The amount of points earned or lost depends on the interactions, and the ability for these interactions to take place depends on the locations of the components involved. In the rest of the paper, we model several rules of the game as a means of illustrating our approach and justifying our claim.

3 Overview of the Architectural Approach

In this section we start by presenting the primitives used for modelling the coordination aspects. Then, we give an overview of the location primitives and explain how global system behaviour emerges, at each state, from the combined effect of the coordination and location contracts that are present in the configuration of the system.

3.1 Coordination concerns

A set of semantic primitives have been put forward in [AF02, AF03] with the aim of supporting an architectural approach to system modelling based on the separation between coordination of interactions among components from the computation of service functionalities within individual components. For instance, in the case of information systems, computations of basic functionalities are usually performed by relatively stable core business entities. Business rules are much more volatile because they have to keep changing for the business to remain competitive. Therefore, they should be modelled separately from the core business services. This clean separation permits the more volatile domain aspects to evolve with minimal impact on core services. Instances of these primitives capture business contracts that can be superimposed dynamically on the relevant core business entities considered as black-boxes [AFLW03].

In order to make precise what we mean by "separation", consider traditional object-oriented modelling (OO). Interactions in OO are established by objects (clients) calling named services of named objects (servers). Such calls are placed during the execution of the methods of the client and, therefore, interactions are not separated from the code that implements the computations. This leads to very tight coupling between components [**?**], not only because any changes on the nature of these interactions requires code to be rewritten, therefore interfering with the computations on the client side, but also because it is very difficult to understand at the conceptual level which interactions are in place at any given state of the system. Another aspect of this tight coupling is the fact that the client needs to know the identity of the server. See [**?**] for a more thorough discussion. The proposed coordination primitives adopt instead an event-base approach [**?**], externalise any interactions among system components as first-class entities and make them explicit in configurations.

Coordination Law : These primitives model rules according to which components can be interconnected. The interconnections are established by what we call coordination

contracts through event-condition-action (ECA) rules. Contracts instantiate laws for particular components that we call *partners*. The components that can become partners of a law are not identified at instance level but by the roles that they are required to play, what we call "coordination interfaces". Auxiliary attributes and operations may also be defined when needed to support the kind of coordination that is required.

Coordination Interface : A coordination interface identifies what is normally called a *role* in Software Architecture. It identifies a class of components in terms of the set of services, events and invariants that need to be provided, either directly or indirectly through refinement, for a component to become coordinated as described by the interaction rules of the law.

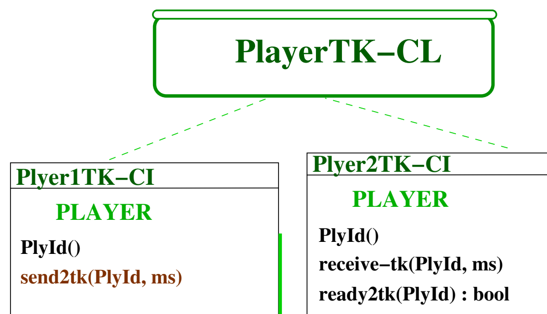
Example 3.1 An elementary interaction between two players is talking with each other. Under this form of interaction, a player decides to initiate a dialogue with another player of his/her choice by broadcasting a message. Depending on the caller and/or the message, the player named in the call can decide to be available for engaging in the dialogue or not. The dialogue is initiated by accepting the message.

```
coordination interface player1TK-CI
partner type PLAYER
datatypes  MESSAGE
events
  send(play:PLAYER,ms:MESSAGE)
end interface
```

```
coordination interface player2TK-CI
partner type PLAYER
datatypes MESSAGE
services
  ready2tk(play:PLAYER,ms:MESSAGE): BOOL
  receive-tk(play:PLAYER,ms:MESSAGE)
end interface
```

Each coordination interface identifies a role. An initiator (an instance of the role `player1TK-CI`) expresses its intention to engage in a dialogue `M` with player `P` by publishing the event `send(P,M)`. A responder (an instance of the role `player2TK-CI`) receives requests through the operation or service `receive-tk`. A responder is also required to provide a service `ready2tk` to indicate with which players and for which messages it is available to engage in a dialogue. The datatype clauses indicate the data sorts and operations that the partners need to have available to interact; in this case, a sort for the messages exchanged.

```
coordination law PlayerTK-CL
partners
        play1:player1TK-CI;
        play2:player2TK-CI
rule  Player's talking
when play1.send2tk(play2,ms)
   with play2.ready2tk(play1,ms)
   do  play2.receive-tk(play1,ms)
end law
```



The coordination rule in this law specifies that `when` the initiator publishes an invitation, the player identified in the event (the responder) decides if it is ready to engage in a dialogue with the initiatior (`with` part). If it is, the responder receives the message. (`do` part).

Even if the example is very simple, it is worth stressing some of the main advantages of externalising interactions. The first advantage is, precisely, what makes the example so simple! Issues such as pricing, message support (SMS, Voice, Multimedia), and so on, can be left to other coordination laws. This allows us not only to concentrate in one aspect at

a time when modelling the application and its domain, but also to evolve them separately. Hence, we may decide to change the way talking is priced or add new message supports without interfering with the protocol followed to engage in a dialogue. The second advantage concerns the distribution dimension. As shown in the next section, this externalisation allows for the rules that determine from which rooms players can talk to other players to be modelled independently of the protocol.

3.2 Distribution Concerns

As emphasised in the introduction, the purpose of location primitives is to enhance architectural mechanisms to deal not only with the coordination of interactions but also with the distribution/mobility dimension. By this we mean aspects usually concerned with the communication infrastructure, including advanced ICT-based elements such as mobile devices, sensors, and so on. However, we wish to support more abstract levels of distribution such as business channels, i.e. dependencies that computations and interactions may have on notions of location that apply to the application domain. This is why we do not work with a fixed notion of location but, instead, provide primitives through which these dependencies can be modelled.

For that purpose, we rely on an abstract specification of a data type that involves a fixed sort LOC for locations and whatever operations are required for modelling the properties of the distribution topology as it applies to the application domain. We can specify subsorts of LOC to model specific kinds of locations such $ROOM$ in our example. Two operations are fixed and used to capture essential aspects for the characterisation of the architectural connectors that handle distribution, what we call location contracts:

- The communication status, i.e the presence, absence, or quality of the link between locations where given services are executing but require data to be exchanged and synchronisation of services to be observed as part of a distributed interaction. This is captured through the "be-in-touch" construct $BT : set(LOC) \rightarrow BOOL$, that is, a boolean operation over locations.

- The ability to continue the execution of an activity at another location, which requires that the new location is reachable from the present one so that the execution context can be moved. This is captured by the construct $REACH : LOC \times LOC \rightarrow BOOL$ that returns whether a given location is reachable from another one.

These primitives have been formalised in CommUnity [LFW02].

As for the coordination dimension, *location laws* are the conceptual modelling primitives through which we capture distribution concerns. We use ECA-like rules triggered by the same kind of events used for coordination laws but, this time, we are concerned with superposing the aspects that are location-dependent. The corresponding *location* interfaces are now concerned with the events and services that characterise the activities from the distribution viewpoint.

Example 3.2 The location-dependent aspects of the interaction that we analised in the previous section, players engaging is a dialogue with another player, concerns the rooms in which the players are acting. Therefore, we need two location interfaces, one for the room of the initiator of the call and one of the room of the responder.

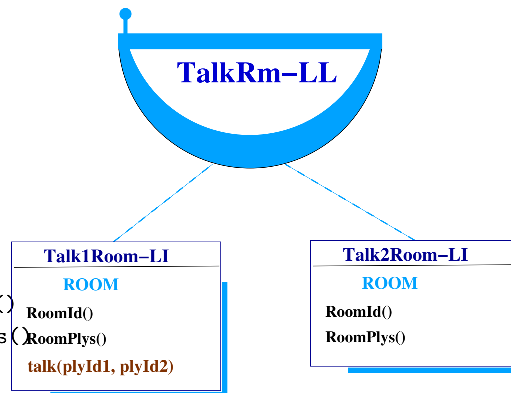
```
location interface Talk1Room-LI
location type ROOM
datatypes
  PLAYER
services
  nbCalls() :  INT;
  maxCalls() :  INT;
  incCalls() :
    post nbCalls() = old nbCalls()+1;
  hasplay(play:PLAYER) : BOOL;
events
  call(play1,play2 :  PLAYER)
end interface
```

```
location interface Talk2Room-LI
location type ROOM
datatypes PLAYER
services
  nbCalls() :  INT;
  maxCalls() :  INT;
  incCalls() :
    post nbCalls() = old nbCalls()+1;
  hasplay(play:PLAYER) : BOOL;
end interface
```

The only difference between the two location interfaces is in the ability to detect calls as events, which only makes sense for the initiator. Otherwise, both interfaces require services that return the number of calls active in the room (`nbCalls`), the maximum number of active calls allowed at any one time (`maxCalls`) and the players currently in the room (`hasplay`), as well as a service that increments the number of calls (`incCalls`).

These two location interfaces are used in the following location law.

```
location law TalkRm-LL
locations
      rm1:  Talk1Room-LI;
      rm2:  Talk2Room-LI;
rule :  Talking from Room(s)
when rm1.call(play1,play2)
    and rm1.hasplay(play1)
    and rm2.hasplay(play2)
  with rm1.nbCalls() < rm1.maxCalls()
    and rm2.nbCalls() < rm2.maxCalls()
    and BT(rm1,rm2)
  do rm1.incCalls and rm2.incCalls
end law
```



The trigger of the location rule (specified under `when`) is any call detected in `rm1` from a player in `rm1` to a player in `rm2`. Calls made in other rooms or when the players are in other rooms are ignored by this law. The trigger is rejected (i.e. the call does not go through) if the number of active calls in either of the rooms has reached the allowed maximum or the rooms are not `in touch`. If the trigger is accepted, both rooms increment the number of active calls.

Notice the use of the operator BT in the `with` clause, rejecting the trigger if the rooms are not `in touch`. Different definitions of BT provide us with different variants of the rules of the game. For instance, in order to set that talking is only allowed within the *same* room, we just have to specify that:

$$BT(rm1, rm2) \Leftrightarrow rm1 = rm2$$

3.3 Integration of concerns

In the two previous subsections, we discussed semantic primitives through which we can separate two different architectural concerns when modelling software systems:

- The mechanisms that should be put in place to coordinate interactions within the system;
- The mechanisms that reflect the dependency of the application domain on a distribution topology that constrains the way computations are performed in locations and interactions take place across locations.

Although there are clear advantages in addressing each of these concerns separately, namely to allow them to evolve independently, it is necessary to bring them together to understand which global system properties emerge at any given state. The fact that we have modelled coordination and location concerns through ECA rules makes it easier to understand how laws compose and which behaviour emerges from their composition. Basically, one has to look for the rules (coordination and location) that are triggered by the same activities.

The joint execution of ECA rules that we have in mind, as formalised in [FLW03] for coordination, takes the conjunction of the guards (`with` clauses) and the parallel composition of the reactions (i.e. the union of the synchronisation sets specified in the `do` clauses). However, in the presence of location rules, we need to take into account that coordination can only be effective when the locations are in touch. That is, when located partners are not in touch, the coordination rules involving the partners do not apply.

Example 3.3 Figure 1 depicts a configuration in which two located players, `play1@rm1` and `play2@rm2` are subject to instances (contracts) of the coordination and location laws discussed in the previous sections. The instantiation of a law over a given set of runtime components requires that the events and services identified in the interfaces of the laws be mapped to runtime activities and actual services provided by the components. For instance, the events `send2tk(play2,ms)` and `call(play1,play2)` required by the coordination interface `player1TK-CI` and the location interface `Talk1Room-LI`, respectively, are both mapped to the same activity of `play1@rm1` that consists in calling `play2@rm2` with a message `ms`.

The reaction to the occurrence of this event depends on whether the locations are in touch or not. If they are, both location and coordination rules apply. Otherwise, only the location rule is activated. In our example, the latter case implies that the guard (`with` clause) of the location rule is false and, hence, the trigger is rejected and nothing happens.

Consider the case in which $BT(rm1, rm2)$ holds, in which case both coordination and location rules apply. The joint guard is the conjunction of the `with` clauses:

```
play2.ready2tk(play1) and rm1.nbCalls() < rm1.maxCalls()
and rm2.nbCalls() < rm2.maxCalls().
```

That is, the call is only allowed to proceed if the responder is willing to talk with the initiator for that particular message, and both rooms can accommodate another call. If this joint guard holds, the reaction is given by the union of the two synchronisation sets:

```
play2.receive-tk(play1,ms) and rm1.incCalls and rm2.incCalls.
```

That is, the responder receives the call and both rooms increment the number of active calls. Note that if the two rooms are the same, there is only one call because both services identified in the location interfaces are instantiated by the same operation `rm.incCalls`.

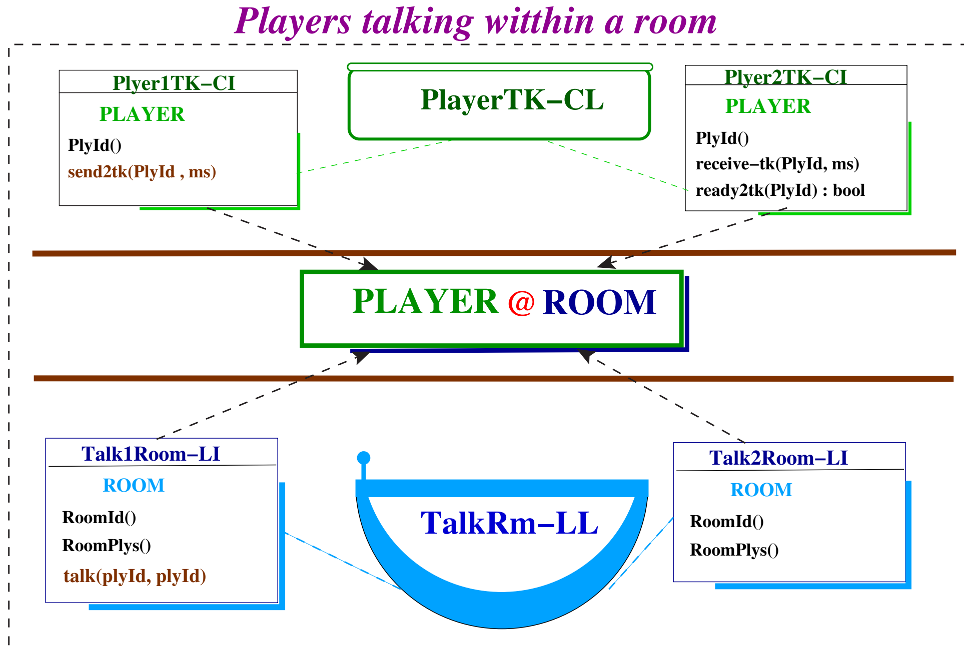


Figure 1: Integration of coordination and location contracts

4 Coordination concerns in the MUD game

In this section, we discuss the coordination dimension in more depth by modelling other rules of the MUD game. Interactions in the game can be classified in two main classes:

Player-player This category includes : (1) players talking to each other as already presented; (2) players moving around (rooms).

Player-object This category concerns: (3) acquisition of (new) objects by a player; (4) release of objects by player; (5) trading objects with other players; (6) using weapons to fight;

We present below the specification of the most representative interactions: trading objects, movement between neighboring rooms and release/acquisition of objects by players. The other forms of interaction are presented in the Appendix.

4.1 Trading objects

Trading involves two players and two objects that have to be in their possession and in a non-broken state. We are going to model the way the exchange is coordinated but not the negotiation process itself. As before, we start by defining the coordination interfaces that specify what is required of each of the partners involved in the interaction.

For the player initiating the trade, we need to know the identity of the objects in its possession (a set denoted `hasObj`). In addition, the event of trading (denoted `trd`) has to be identified and a service for capturing the result of the exchange of objects (denoted by `exchg`) has to be provided.

```
coordination interface player1TrdObj-CI
partner type PLAYER
Datatypes
  OBJECT;
services
  hasObj(obj:OBJECT) : BOOL;
  acquire(obj:OBJECT)
     pre ¬hasObj(obj)
     post hasObj(obj);
  release(obj:OBJECT)
     pre hasObj(obj)
     post ¬hasObj(obj);
events
  trd(playId:PLAYER,obj1,obj2:OBJECT)
end interface
```

Notice that the purpose of the coordination interfaces is to specify what events and services are required of the partners, not how they are provided. The "how" needs to be made explicit during instantiation, which is performed at run-time, by binding these event and service declarations to what the run-time components provide. This means that the binding mechanisms may require additional computations from the more basic services on offer. For instance, checking that an object is in possession of a player depends on the way the ownership relation is represented in the system. It does not make sense to commit for a particular representation at the conceptual modelling level.

Also note that we can specify requirements on the services through pre/post-conditions. Again, it does not make sense to specify how these specifications are implemented.

The same features are required of the second player except for the event through which the trade is initiated. Instead, a service for accepting the trade (denoted by `accept`) is required.

```
coordination interface player2TrdObj-CI
partner type PLAYER
Datatypes
  OBJECT;
services
  hasObj(obj:OBJECT) : BOOL;
  acquire(obj:OBJECT)
     pre ¬hasObj(obj)
     post hasObj(obj);
  release(obj:OBJECT)
     pre hasObj(obj)
     post ¬hasObj(obj);
  accept(playId:PLAYER,obj1,obj2:OBJECT)
end interface
```

To be sure that the objects being traded are not broken, we are going to define a coordination interface that will allow objects to be directly involved in the law as partners:

```
coordination interface ObjectTrded-CI
partner type OBJECT
services   broken() :  BOOL
end interface
```

Again, we refrain from making explicit how the fact that object is broken is derived from

its state. This has to be provided through the binding mechanisms only at run-time through instantiation.

The law that coordinates trading objects can now be defined:

```
coordination law TradeObjs-CL
partners
      play1:player1Trdo-CI;
      play2:player2Trdo-CI
      obj1,obj2:  ObjectTrded-CI
rule  Trade-objects
when play1.trd(play2,obj1,obj2)
 with ¬obj1.broken() and ¬obj2.broken() and
    play2.accept(play1,obj2,obj1) and
    play1.hasObj(obj1) and play2.hasObj(obj2)
 do play1.acquire(obj2) and
     play2.acquire(obj1) and
     play1.release(obj1) and
     play2.release(obj2)
end law
```

The trigger of this coordination law is the event through which player `play1` proposes player `play2` to trade `obj1` for `obj2`. The trigger is refused if the objects do not belong to the players, the second player does not accept the trade, or any of the objects is broken. Otherwise, the exchange takes place.

4.2 Players acquiring/releasing objects

The coordination interfaces `player1Trdo-CI` and `player2Trdo-CI` defined above request services that account for acquisition and release objects, which are used in the reaction to the trigger of the law to perform an exchange. As already mentioned, the reaction to the trigger is performed as an atomic transaction. Therefore, it may fail if, for some reason, one the acquisitions or releases is not enable. Indeed, other coordination rules may apply to one of these operations that depends on the nature of the object or the player. This is how we can model rules of the game that apply to certain classes of players and objects.

Just to give a very simple example, consider a rule that restricts objects of weight greater than 100 to be acquired by players with strength greater than 10:

```
coordination interface playerStrength-CI
partner type PLAYER
datatypes OBJECT
services
  strength() :  [1..12];
events
  acquire(obj:OBJECT)
end interface
```

```
coordination interface objectWeight-CI
partner type OBJECT
services   weight() :  NAT
end interface
```

```
coordination law acqStreWei-CL
partners
      play:  playerStrength-CI;
      obj:  objectWeight-CI
rule
when play.aquire(obj)
  with (obj.weight() ≥ 100) or play.strength() ≥ 10    do true
end law
```

The coordination rule blocks the acquisition if the object has weight greater than 100 and the player has strength less than 10.

4.3 Moving between rooms

Although, at first glance, the movement of players between rooms seems to be of a pure distribution concern, the following observations demonstrate that it concerns coordination as well. For instance, if a player moves to the next level, it has to release all objects in his possession. If a player moves to a neighbouring room, the objects follow the player but have to be registered in the new room. One may think of changing these rules to define variants of the game without having to change the way the movement takes place in terms of locations.

Another example can be given in terms of a player's death. The rules of the game concern both distribution (the dead player moves to the starting room) and coordination (all objects need to be released). One may well think of changing one dimension (e.g. allowing the dead player to keep objects of a given class) without changing the other (e.g. the fact that the dead player goes back to the start).

In summary, it makes sense to separate the two concerns. We consider below only the movement between neighbouring rooms. The other two cases are treated in the appendix.

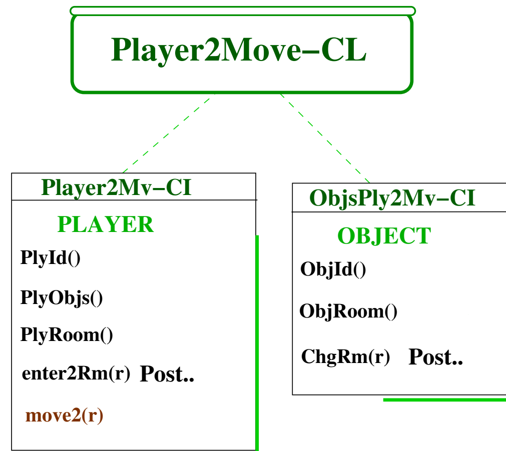
As already mentioned, the coordination aspects involved in a movement between rooms concerns the interactions between players and objects. This gives rise to a coordination law for pairs player-object triggered by requests from players to move to another room. This law requires two coordination interfaces, one for the player and one for the object, each of which offers services for the move to be effected. The player is also required to publish the request to move as an event.

```
coordination interface player2Mv-CI
partner type PLAYER
Datatypes OBJECT, ROOM;
services
  hasObj(obj:OBJECT) : BOOL;
  playRoom() :  ROOM;
  enter2Rm(r:ROOM)
     post playRoom = r
events
  move2(r:ROOM)
end interface
```

```
coordination interface Objsplay2Mv-CI
partner type OBJECT
Datatypes ROOM;
services
  ObjRoom() :  ROOM;
  ChgRm(r:ROOM)
     post ObjRoom() = r
end interface
```

The corresponding coordination law is as follows:

```
coordination law Player2Move-CL
partners
    play:Player2Mv-CI;
    obj:Objsplay2Mv-CI
rule Mvt-player
when play.move2(r) and play.hasObj(obj)
       do obj.ChgRm(r) and
  play.enter2Rm(r)
end law
```



It is important to stress that coordination laws are *types* in the sense that they model generic forms of interaction. Hence, the law above concerns any pair player-object, i.e. any instances of the declared coordination interfaces. However, any instance of the law is only triggered for players that publish a request to move and the objects that they have in their possession. That is, at run-time, only the objects belonging to a player are concerned when that player requests to move.

Furthermore, the trigger may be refused if, for some reason, either the player or the object cannot move. For instance, one may wish to impose restrictions on certain objects to move to certain rooms, say by virtue of size or substance. In this case, the player will not be able to move until the object is released. Indeed, as already explained, the superposition of contracts that share the same trigger required the synchronisation of their reactions. Hence, it is enough that one of the object of the player cannot move for the player not to be able to move.

Notice that no considerations are made as to the nature of the room as a location; such information is to be considered in the location laws that apply to that trigger.

5 Location concerns in the MUD game

We have already justified why we take rooms to provide the locations of the distribution dimension. In this section, we illustrate how this dimension needs to be addressed to provide a correct model of the game as captured by its rules. More precisely, we focus on the rules used in the previous section to illustrate the modelling of coordination concerns. A more complete account is left to the appendix.

5.1 Players' trading

Principally, the main ideas we already presented about location concerns for talking are to be applied for the trading action. The only difference here is that we have two cases (trading with objects and trading with money), and that the traded objects should reside with their respective players in the same room. As we pointed out, the trading with money is specified in the appendix.

5.1.1 Trading objects

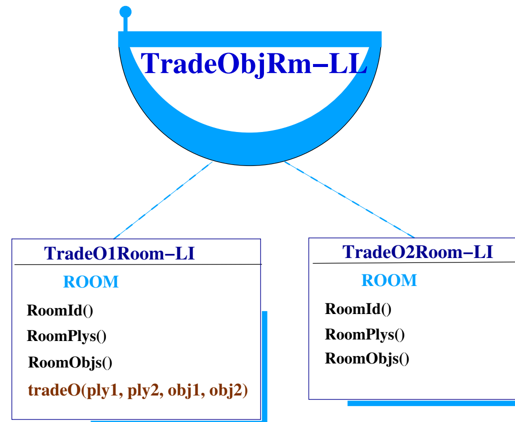
From the location perspective, for trading objects between players, each player and its corresponding to-be-traded object should be in the same room. As we have done for the talking, we let open the fact that both players have to be in the same room, so that different variant could be conceived while precisely stating the definition of the "be-in-touch" relationship. in other words, we consider two room location interfaces.

```
location interface Trade01Room-LI
location type ROOM
datatypes
  ROOM-ID, PLAYER-ID, OBJECT-ID
services
  RoomId() : ROOM-ID;
  Roomplays() : List[PLAYER-ID];
  RoomObjs() : List[OBJECT-ID];
events
  tradeO(play1, play2 : PLAYER-ID,
      obj1, obj2: OBJECT-ID)
end interface
```

```
location interface Trade02Room-LI
location type ROOM
datatypes
  ROOM-ID, PLAYER-ID, OBJECT-ID
services
  RoomId() : ROOM-ID;
  Roomplays() : List[PLAYER-ID];
  RoomObjs() : List[OBJECT-ID];
end interface
```

In the corresponding location law, two object identifiers are required each assumed to be in the possession of one of two players. Besides the communication status that should be true within/between such room(s), i.e. **BT**(rm1.RoomId(),rm2.RoomId()), the objects have to belong to the room(s) of their respective players. Please be aware finally that the fact that the objects should be in the possession of the respective players is not a location concerns, but instead an interaction concerns we already addressed in the respective coordination law. That is, it is only at the integration phase that all these functionalities/constraints have to be merged to reflect the correct meaning of the action.

```
location law TradeObjRm-LL
locations
      rm1: Trade01Room-LI;
      rm2: Trade02Room-LI;
rule : Talk from Room(s)
when rm1.tradeO(play1,play2, obj1, obj2)
      and BT(rm1.RoomId(),rm2.RoomId())
      with (play1 ∈ rm1.Roomplays()) and
           (play2 ∈ rm2.Roomplays()) and
           (obj1 ∈ rm1.RoomObjs()) and
           (obj2 ∈ rm2.RoomObjs())
      do return true
when rm1.tradeO(play1,play2, obj1, obj2)
      and ¬ BT(rm1.RoomId(),rm2.RoomId())
      with false
      do return false
end law
```



5.2 Player's Moving

The movement of players involves two rooms, which should have doors to each other. This rule concerns a "normal" move between two neighboring rooms. That is why we have also

to consider the movement of a dead player, which is from any room to the starting room in the same level. The movement to a next level is also slightly different from these two ones. These two later kind of movement are specified in the appendix.

5.2.1 Neighboring Movement

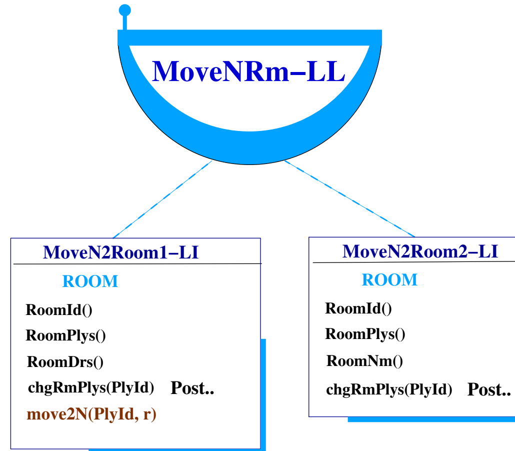
All what is required for such a move is the list of players (identities) from the concerned rooms and the doors for each room. For a given room, the doors can simplay be specified by the list of accessible rooms (identities).

```
location interface Move2Room1-LI
location type ROOM
datatypes
  ROOM-ID, PLAYER-ID
services
  RoomId() : ROOM-ID;
  Roomplays() : List[PLAYER-ID];
  RoomDrs() : List[ROOM-ID];
  chgRmplays(play:PLAYER-ID)
    post remove(play, Roomplays())
events
  move2N(play:PLAYER-ID, r:ROOM-ID)
end interface
```

```
location interface Move2Room2-LI
location type ROOM
datatypes
  ROOM-ID, PLAYER-ID, ROOM-NAME
services
  RoomId() : ROOM-ID;
  Roomplays() : List[PLAYER-ID];
  RoomNm() : List[ROOM-NAME];
  chgRmplays(play:PLAYER-ID)
    post add(play, Roomplays())
end interface
```

The service `chgRmplays` allows updating the list of the players in the room after a player has left or entered a new room. The location law calls these services after it checks the existence of a door between them. We have added the room name (e.g. "starting", "special", etc) to prevent any player to be back to a starting room even if there is a door leading to it—as imposed by the game.

```
location law MoveNRm-LL
locations
        rm1: MoveN2Room1-LL;
        rm2: MoveN2Room2-LL;
rule : Normal room moving
when rm1.Move2N(play, r)
  and REACH(rm1.RoomId(),rm2.RoomId())
  with (play ∈ rm1.Roomplays()) and
    (rm2.RoomId() ∈ rm1.RoomDrs()) and
    (rm2.RoomNm() ≠ "starting") and
    (rm1.RoomNm() ≠ "special")
  do rm1.chgRmplays(play) and
    rm2.chgRmplays(play)
when rm1.Move2N(play, r)
  and ¬ REACH(rm1.RoomId(),rm2.RoomId())
  with false
  do return false
end law
```



Finally is very relevant to imprecise that, as we are dealing with a movement, in this location law a reachability between the rooms is required rather than a communication as it was the case for talking and trading. The second important observation concerns the specification of the reachability itself. In fact, the reachability $REACH(rm_1, rm_2)$ between

the rooms rm_1 and rm_2 is equivalent to the existence of a door from rm_1 to rm_2. We have to check the following equivalence:

$$REACH(rm_1, rm_2) = true \Leftrightarrow rm_2.RoomId() \in rm_1.RoomDrs.$$

5.3 Player's acquiring/releasing objects

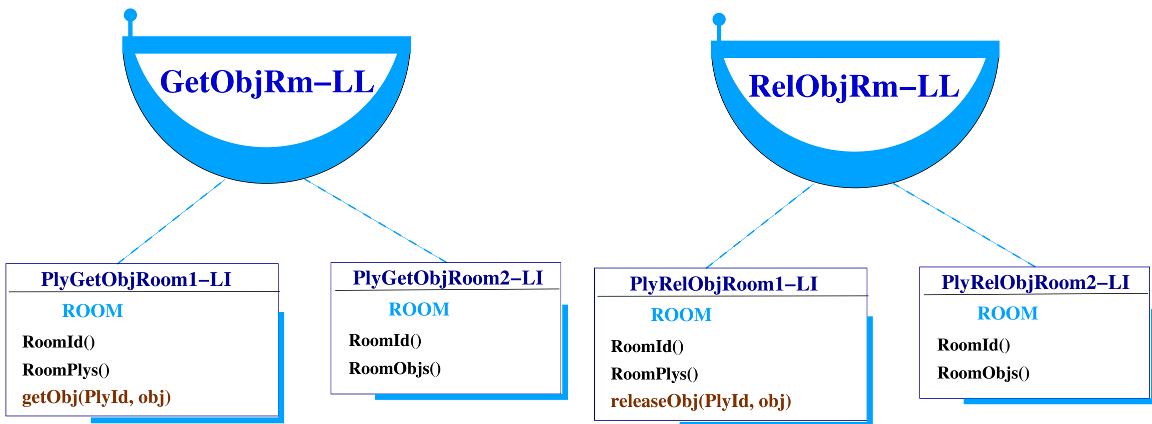
The location imposes that players acquire and release objects only within a same room. Nevertheless, for the same raison invoked in the players talking location modelling, we use two interfaces to allow a direct extension of the game rules where players in given room could acquire/release objects from another "in touch" room.

```
location interface playGetObjRoom1-LI
location type ROOM
datatypes
  ROOM-ID, PLAYER-ID, OBJECT-ID
services
  RoomId() :  ROOM-ID;
  Roomplays() :  List[PLAYER-ID];
events
  getObj(play:PLAYER-ID,
      obj:OBJECT-ID)
end interface
```

```
location interface playRelObjRoom1-LI
location type ROOM
datatypes
  ROOM-ID, PLAYER-ID, OBJECT-ID
services
  RoomId() :  ROOM-ID;
  Roomplays() :  List[PLAYER-ID];
events
  releaseObj(play:PLAYER-ID,
      obj:OBJECT-ID)
end interface
```

```
location interface playGetObjRoom2-LI
location type ROOM
datatypes
  ROOM-ID, PLAYER-ID, OBJECT-ID
services
  RoomId() :  ROOM-ID;
  RoomObjs() :  List[OBJECT-ID];
end interface
```

```
location interface playRelObjRoom2-LI
location type ROOM
datatypes
  ROOM-ID, PLAYER-ID, OBJECT-ID
services
  RoomId() :  ROOM-ID;
  RoomObjs() :  List[OBJECT-ID];
end interface
```



In contrast to the coordination perspective, in the location laws below, for acquiring/releasing objects by players we have just to check the room(s) where these objects and players are residing are in touch, that is:

(1) **BT**`(rm1.RoomId(),rm2.RoomId())=True`; and

(2) `(play ∈ rm1.Roomplays())` **and** `(obj ∈ rm2.RoomObjs())`

In such positive case, we just return a `true` indicating that the acquire/release is authorized.

```
location law GetObjRm-LL
locations rm1:  playGetObjRoom1-LL;
        rm2:  playGetObjRoom1-LL;
rule :  Acquiring Objects
when rm1.GetObj(play, obj)
  and BT(rm1.RoomId(),rm2.RoomId())
  with (play ∈ rm1.Roomplays()) and
    (obj ∈ rm2.RoomObjs())
  do retrun true
when rm1.getObj(play, obj)
  and ¬ BT(rm1.RoomId(),rm2.RoomId())
  with false
  do return false
end law
```

```
location law RelObjRm-LL
locations rm1:  playRelObjRoom1-LL;
        rm2:  playGetObjRoom1-LL;
rule :  Acquiring Objects
when rm1.releaseObj(play, obj)
  and BT(rm1.RoomId(),rm2.RoomId())
  with (play ∈ rm1.Roomplays()) and
    (obj ∈ rm2.RoomObjs())
  do retrun true
when rm1.releaseObj(play, obj)
  and ¬ BT(rm1.RoomId(),rm2.RoomId())
  with false
  do return false
end law
```

We again recall that, for the considered variant of the game, the existence of communication is restricted to same room, that is:

$$BT(rm_1, rm_2) = true \Leftrightarrow rm_1.RoomId() = rm_2.RoomId()$$

6 Integration of Concerns in the MUD game

After specifying each of the involved interaction and location concerns in terms of coordination and location laws respectively, the next step is to accordingly bring together these concerns around each game action. In this way and only after this integration the complete functionality of each action is correctly captured.

Subsequently we go therefore through the already specified actions and integrate their coordination laws and respective location laws. This integration consists in: (1) unifying the (coordination and location) events triggering the action and (2) synchronizing the different parts of the law accordingly (i.e. a conjunction of the `with` and `do` parts of the involved coordination and location laws when the BT or REACH holds).

6.1 The player talking action at rooms

Although we have already addressed the integration of concerns with respect to this action, in the following we want just to emphasize the fact that the modeling of this case study through architectural techniques and more specifically interaction and location primitives, presents many advantages and flexibility compared to the modeling of the same case study using for instance the KLAIM approach (i.e. KLAVA implementation) [BDNP02] or UML [BJR98] as it has been achieved by our project partners in Pisa and Munich.

Indeed, by separating both concerns and integrating them at runtime we easily adapt them when the game rules change. As illustration of this fact is the possibility of dealing with the "real" (i.e. physical) player locations (i.e. mobile phones). In such case, as depicted in Figure 2 we have just to adapt the location law by introducing two corresponding new locations (i.e. `LOC1` and `LOC2`) and consider the room just for registering the players. In the location law we then require that $BT(rm, loc_1, loc_2)$ holds, that is, (from his/her mobile) a player can communicate in the room where he is registered and with other players for talking (trading or fighting).

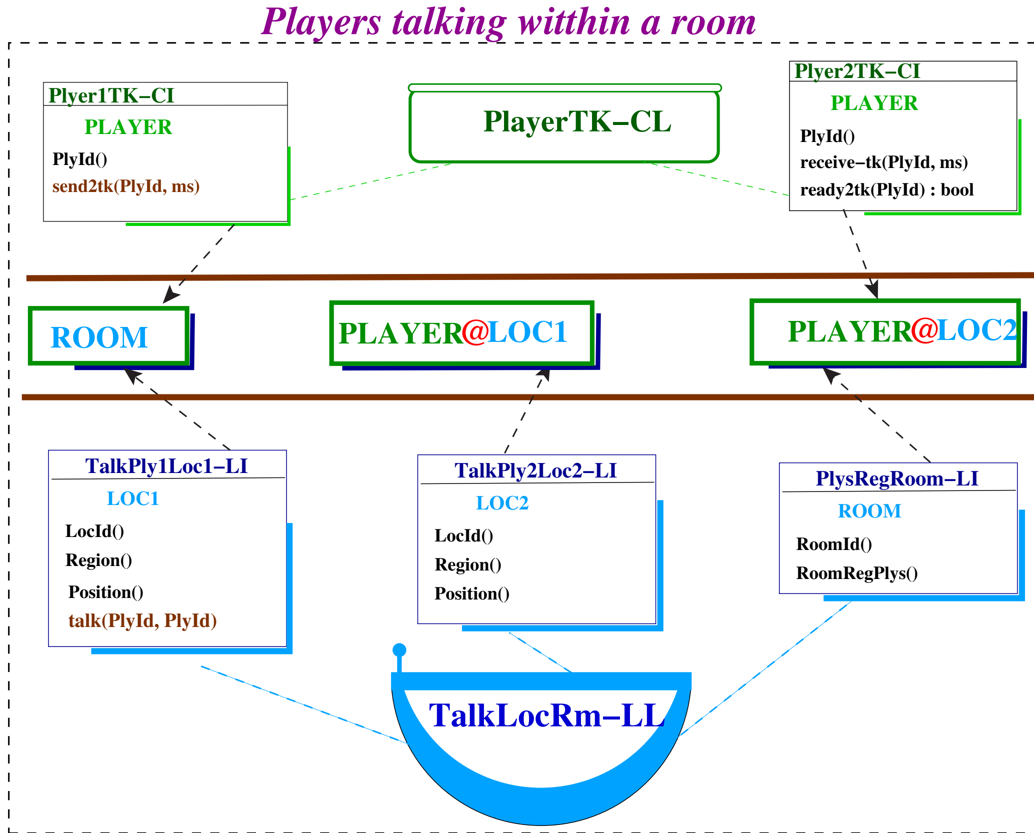


Figure 2: The C/L-based talking action with different players locations

6.2 Trading action at rooms

As we have developed, the trading could be either using objects or by exchanging objects with money. The integration of concerns associated with the first case is depicted in the following Figures 3. As the synchronization of the corresponding could easily be achieved in the way as above, we have just brought the two laws to stress the need for integrating them.

```
coordination law PlayerTRDO-CL
partners
  play1:player1Trdo-CI;
  play2:player2Trdo-CI
  objplay1,objplay2:  ObjectTrded-CI
rule  Trade-objects
when play1.trd(play2.playId(),obj1)
 with (objplay1.ObjState()
  ∧ objplay2.ObjState() ≠ "broken")
  and objplay1.ObjAssign() = play1.playId()
  ∧ objplay2.ObjAssign() =play2.playId())
 do play1.exchg(obj1, objplay2.ObjId())
  and play2.exchg(objplay2.ObjId(), obj1)
  and objplay1.traded(play2.playId())
  and objplay2.traded(play1.playId())
end law
```

```
location law TradeORm-LL
locations
  rm1:  TradeO1Room-LI;
  rm2:  TradeO2Room-LI;
rule :  Talk from Room(s)
when rm1.tradeO(play1,play2, obj1, obj2)
  and BT(rm1.RoomId(),rm2.RoomId())
  with (play1 ∈ rm1.Roomplays()) and
    (play2 ∈ rm2.Roomplays()) and
    (obj1 ∈ rm1.RoomObjs()) and
    (obj2 ∈ rm2.RoomObjs())
  do return true
when rm1.tradeO(play1,play2, obj1, obj2)
  and ¬ BT(rm1.RoomId(),rm2.RoomId())
  with false
  do return false
end law
```

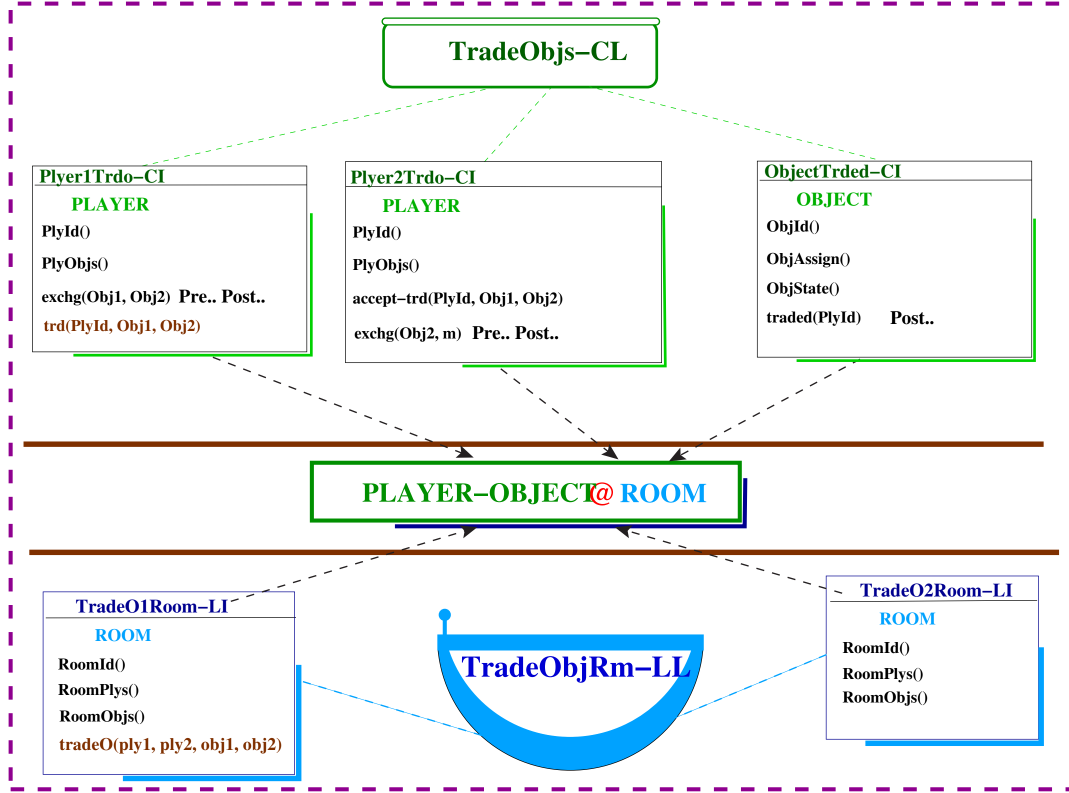


Figure 3: The C/L-based object trading action at rooms

6.3 The player moving action at rooms

As we have developed, the moving could be either to a neighboring, to the starting or to a next level. The integration of concerns for the first case is depicted in the following Figures 4. Their associated coordination and location laws could be recalled as below:

```
coordination law playMv2Rm-CL
partners
  play:PlayerMv2Rm-CI;
  Objsplay[|play.playObjs|]:
    Objsplay2Release-CI
rule  Mvt-player
when play.move2(r)
 with
   for i:= 1 to |play.playObjs|
    (Objsplay[i].ObjId() ∈ playObjs)
   do Objsplay[i].ChgRm(r)
 play.enter2Rm(r)
end law
```

```
location law MoveNRm-LL
locations rm1:  MoveN2Room1-LL;
          rm2:  MoveN2Room2-LL;
rule :  Normal room moving
when rm1.Move2N(playId, r)
  and REACH(rm1.RoomId(),rm2.RoomId())
  with (playId ∈ rm1.Roomplays()) and
    (rm2.RoomId() ∈ rm1.RoomDrs()) and
    (rm2.RoomNm() ≠ "strating")
  do rm1.chgRmplays(play) and
    rm2.chgRmplays(play)
when rm1.Move2N(playId, r)
  and ¬ REACH(rm1.RoomId(),rm2.RoomId())
  with false
  do return false
end law
```

Their synchronisation in the case of the availability of reachability between the two concerned rooms, starts by unifying their triggering to a single common. We denote such event

as play@rm1.Move2N(play, r) (i.e. the one from the location as it contains all information). The synchronisation itself could be described as follows:

```
Normal moving between rooms
when play@rm1.Move2N(play, r)
  and REACH(rm1.RoomId(),rm2.RoomId())
  with (playId ∈ rm1.Roomplays()) and (rm2.RoomId() ∈ rm1.RoomDrs())
    and (rm2.RoomNm() ≠ "strating")
    and for i:= 1 to |play.playObjs| (Objsplay[i].ObjId() ∈ playObjs)
  do rm1.chgRmplays(play) and rm2.chgRmplays(play)
    and for i:= 1 to |play.playObjs| (Objsplay[i].ChgRm(r) and play.enter2Rm(r)
end synchronisation
```

That is, by receiving the movement event, and under the availability of reachability between the two rooms, first we have to check that there is a door between the two concerned rooms (i.e. they are neighboring rooms) and that we are not going back to the starting room. This test concerns the location concern. On the coordination side, we have to check that all the object (identity) instances are selected to be moved with the player. The do part also deal with both concerns. That is, from the location side, we have to update the two rooms' players list. From the coordination side, we have to update the player's objects location, with the new room, and finally update the player room.

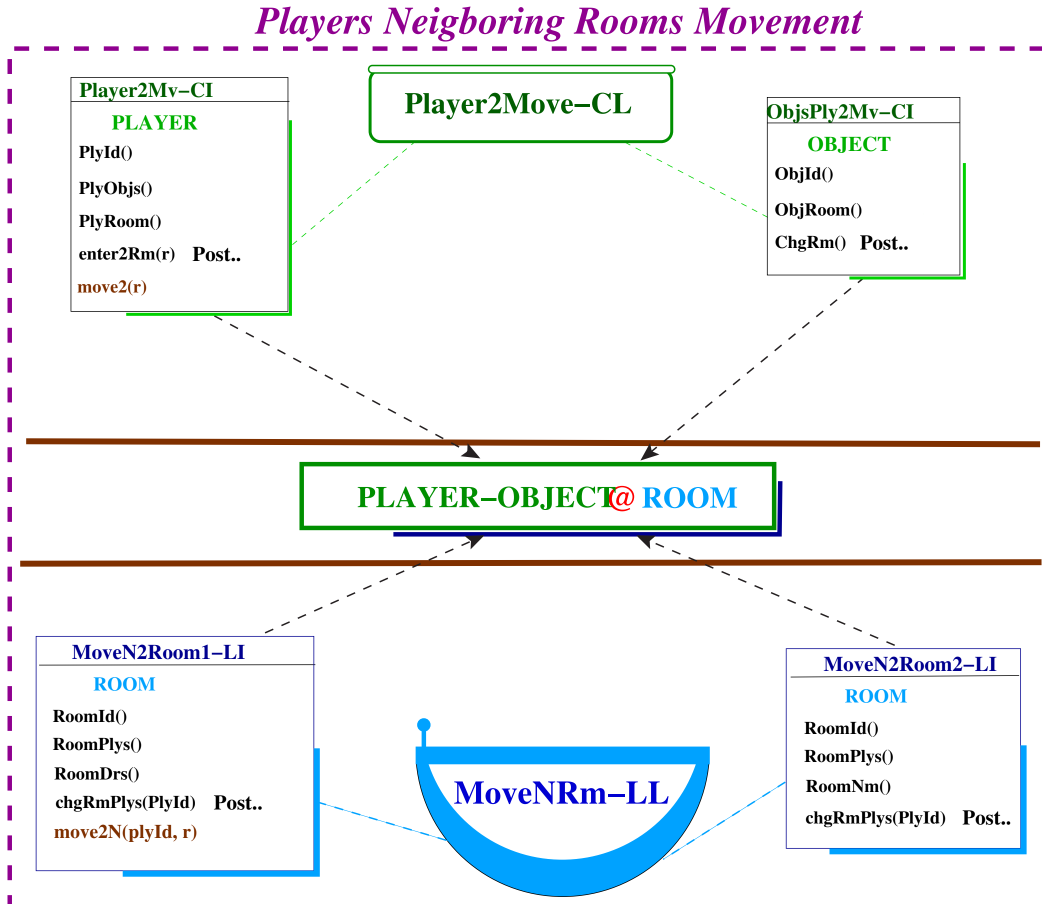


Figure 4: The C/L-based neighboring moving action at rooms

6.4 The players acquiring/releasing objects action at rooms

In the same way we bring together the associated coordination and location law to correctly reflect these two related actions. We depict them in the following Figures 5.

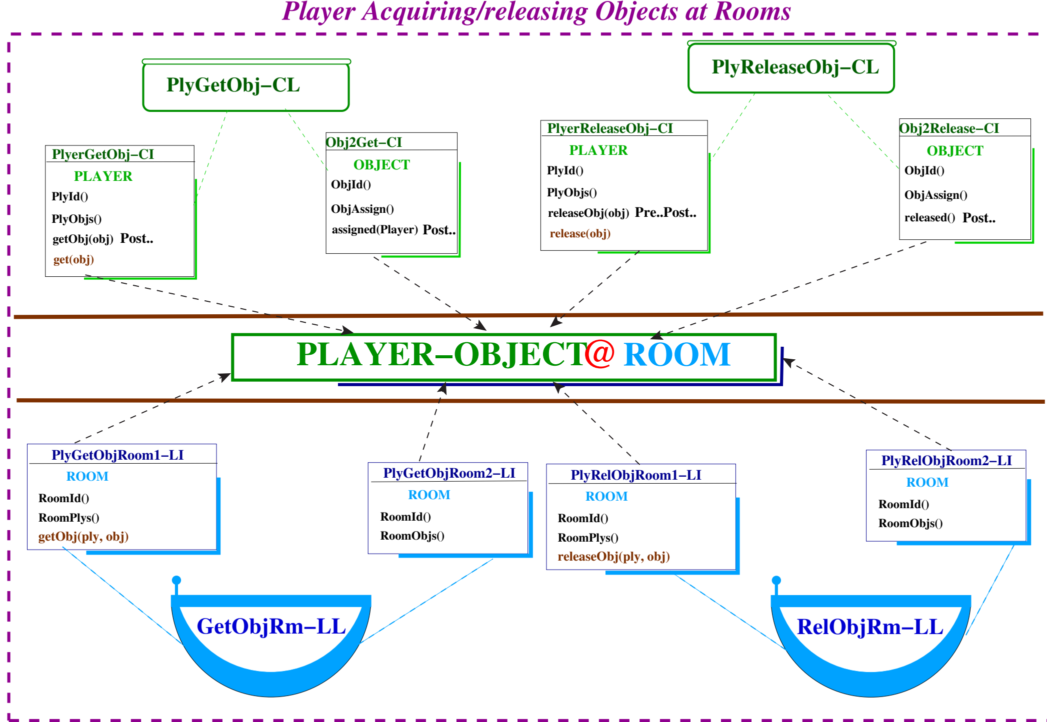


Figure 5: The C/L-based acquring/releasing action at rooms

7 Conclusions

In this paper we put forward a dynamically adapted specification of the MUD game through our architectural conceptual primitives of location and coordination laws. We also showed how both concerns are separately specified and brought together to capture the correct functionality of each action in the game. The main benefits of our approach reside in its intrinsic flexibility by allowing different variants of the game to be easily adopted. It also demonstrates that game descriptions are in general intrinsically rule-based oriented and should therefore be so in their specification/implementation.

The benefits of tackling this particular case study are manyfold. First, through this application we show how interaction and location concerns are naturally specified/adapted in a separate and independent way following the game rules associated with each concern. Second, in order to correctly reflect the functionality and meaning of each action/step of the game, these concerns are to be integrated by unifying the triggering events in each concern. Third, we demonstrate how the externalization of these interaction and distribution/mobility concerns at the architectural level results in very simple and flexible specification, where different variants of the game can easily be conceived—without resorting to changing the computational components which become very minimal and secondary.

In the near future we are planning to implement this detailed specification within the

location environment that our ATX colleagues are developing by extending the CDE environment that deals just with coordination [AGKF02]. Another perspective consists in implementing this conceptual solution directly in the CommUnity Workbench [OW04] using the distributed tuple-based implementation of KLAVA [BDNP02] environment. Besides that we are also working on implementing this case study into the Maude language [CDE+99] following the axiomatization we recently put forward for the approach using tailored rewrite theories[AF05], in Meseguer's rewrite logic [Mes92].

ASPECTS

References

[ABB+03] L. Andrade, P. Baldan, H. Baumeister, R. Bruni, A. Corradini, R. De Nicola, J. 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 architecture for mobility. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *WADT*, volume 2755 of *Lecture Notes in Computer Science*, pages 1–33. Springer, 2003.

[AF02] L. Andrade and J. Fiadeiro. Agility through coordination. *Information Systems*, 27:411–424, 2002.

[AF03] L. Andrade and J. Fiadeiro. Architecture Based Evolution of Software Systems. In M.Bernardo and P.Inverardi, editor, *Proc. of Formal Methods for Software Architectures*, volume 2804 of *Lecture Notes in Computer Science*, pages 148–181. Springer, 2003.

[AF05] N. Aoumeur and J. Fiadeiro. Architectural Specification of Location-aware Systems in Rewriting logic. Technical Report, 2005. *Submitted for Publication.*

[AFLW03] L. Andrade, J. Fiadeiro, A. Lopes, and M. Wermelinger. Coordination for Distributed Business Systems. In R.Mittermeir J.Eder and B.Pernici, editors, *Proc. of Information Systems for a Connected Society*, pages 27–37. University of Maribor Press, 2003.

[AFO04a] N. Aoumeur, J. Fiadeiro, and C. Oliveira. Distribution Concerns in Service-Oriented Modelling. In S.Weerawarana, editor, *2nd International Conference on Service Oriented Computing (ICSOC'04), Short papers*, IBM REsearch Divison, IBM Report:RA221 (W0411-084), pages 26–35, 2004.

[AFO04b] N. Aoumeur, J. Fiadeiro, and C. Oliveira. Towards an Architectural Approach for Location-aware Business Processes. In *Proc. of the 13th IEEE International Workshops on Enabling, Technologies : Infrastructure for Collaborative Enterprises, June 14-16*, pages 147–152. IEEE Computer Society, 2004.

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[AGKF02] L. Andrade, J. Gouveia, G. Koutsoukos, and J. Fiadeiro. Coordination Contracts, Evolution and Tools. *Software Maintenance and Evolution: Research and Practice*, 14(5):353–369, 2002.

[BBH+04] M. Barth, H. Baumeister, F. Hacklinger, P. Kosiuczenko, and A. Rauschmayer. Instructions for swepmud. Technical report, Universität Muechen, Germany, 2004.

[BDNP02] L. Bettini, R. De Nicola, and R. Pugliese. Klava: a java framework for distributed and mobile applications. *Software - Practice and Experience*, 32(14), 2002.

[BJR98] G. Booch, I. Jacobson, and J. Rumbaugh, editors. *Unified Modeling Language, Notation Guide, Version 1.0.* Addison-Wesley, 1998.

[CDE+99] M. Clavel, F. Duran, S. Eker, J. Meseguer, and M. Stehr. Maude : Specification and Programming in Rewriting Logic. Technical report, SRI, Computer Science Laboratory, March 1999. URL : http://maude.csl.sri.com.

[Fia04] J. Fiadeiro. *Categories for Software Engineering.* Springer, 2004.

[FL04] J. Fiadeiro and A. Lopes. CommUnity on the Move: Architectures for Distribution and Mobility. In *Formal Methods for Components and Objects*, volume 3188 of *Lecture Notes in Computer Science*, pages 177–198. Springer, 2004.

[FLW03] J. Fiadeiro, A. Lopes, and M. Wermelinger. A mathematical semantics for architectural connectors. In *Generic Programming*, volume 2793 of *Lecture Notes in Computer Science*, pages 190–234. Springer, 2003.

[LFW02] A. Lopes, J.L. Fiadeiro, and M Wermelinger. Architectural primitives for distribution and mobility. In *Proc. of ACM SIGSOFT Symp. on Foundations of Software Eng.*, pages 41–50. ACM Press, 2002.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model for concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[OW04] C. Oliveira and M. Wermelinger. The CommUnity workbench. In *Proc. of the 26th Intl. Conf. on Software Engineering*, pages 709–710. IEEE Computer Society Press, 2004.

[WKA+03] M. Wermelinger, G. Koutsoukos, R. Avillez, J. Gouveia, L. Andrade, and J. Fiadeiro. Using Coordination Contracts for Flexible Adaptation to Changing Business Rules. In *Proc. of the 6th Intl. Workshop on Principles of Software Evolution*, pages 115–120. IEEE Computer Society Press, 2003.

A Interaction concerns : Continuation

A.1 Object trading

A.1.1 Trading with Money

In contrast to the trading of object/object, the trading object/money involves just one object (assigned to the player triggering the trade) and money from both players.

```
coordination interface player1Trdm-CI
partner type PLAYER
Datatypes
  PLAYER-ID, OBJECT-ID, MONEY;
services
  playId(): PLAYER-ID;
  playObjs() : List(OBJECT-ID);
  Money() : MONEY;
  Exchg(obj:OBJECT-ID, m:MONEY)
    pre obj ∈ playObjs()
    post remove(obj, playObjs()) and Money+ = m;
events
  trd(play2:PLAYER-ID; obj:OBJECT-ID, m:MONEY)
end interface
```

```
coordination interface player2Trdo-CI
partner type PLAYER
Datatypes
  PLAYER-ID, OBJECT-ID;
services
  playId(): PLAYER-ID;
  playObjs() : List(OBJECT-ID);
  Money() : MONEY;
  Exchg(obj:OBJECT-ID, m:MONEY)
    pre obj ∉ playObjs()
    post add(obj, playObjs()) and Money- = m;
  accept-trd(play1:PLAYER-ID, obj:OBJECT-ID,
          m:MONEY): Boolean
end interface
```

```
coordination interface ObjectTrdo-CI
partner type OBJECT
Datatypes
  PLAYER-ID, OBJECT-ID;
services
  ObjId(): OBJECT-ID;
  ObjAssign() : PLAYER-ID;
  ObjState() : OBJECT-STATE;
  traded(play:PLAYER-ID) post ObjAssign() := play
end interface
```

The requirements for trading objects with money is the same as trading object/object except that now the second player has to pay money for the object. That is, the two services *Exchg* allow now for exchanging the traded object with money.

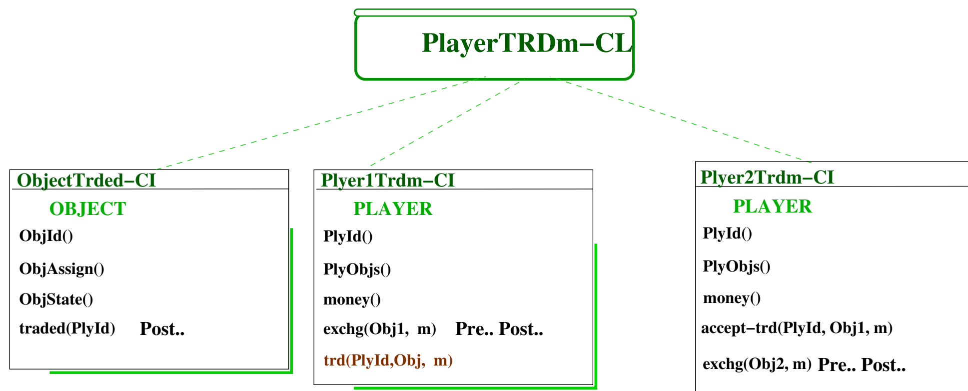


Figure 6: The Object/Money Trading coordination law

```
coordination law PlayerTRDM-CL
partners play1:player1Trdo-CI; play2:player2Trdo-CI
        objplay1 : ObjectTrdo
rule Trade-Money
when play1.trd(play2.playId(),obj, m)
  with (objplay1.ObjState() ≠ "broken") and
    (objplay1.ObjAssign() = play1.playId())
```

```
 do play1.exchg(obj, m) and
   play2.exchg(obj, m) and
   objplay1.traded2(play2.playId())
end law
```

A.2 Object moving

A.2.1 Moving Dead Player

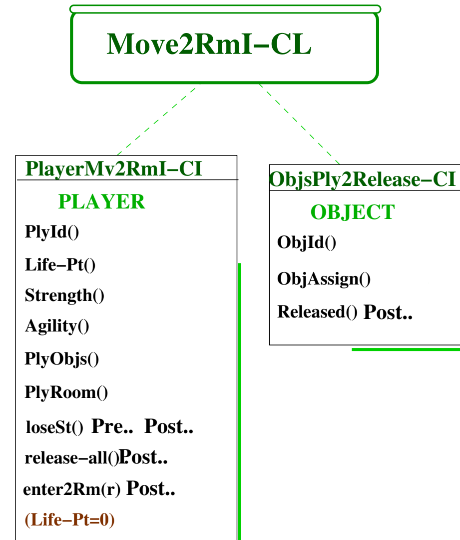
As stated in the game description a dying player must be brought to the initial starting room of that level. Moreover, his points such as agility and strength are to initialized to default initial values, which we denote respectively by `Dfagl` and `Dfstg`. Besides that, in contrast to neighboring move we already conceived, on the one hand, a dying player has to release all his objects before being moved to the initial room. On the other hand, the event triggering this move is a proactive one and corresponds to the life points reaching zero (i.e. declared dead).

```
coordination interface playerMv2RmI-CI
partner type PLAYER
Datatypes
  PLAYER-ID, OBJECT-ID, ROOM-ID;
services
  playId():  PLAYER-ID;
  Life-Pt() :  Natural;
  Strength() :  Natural;
  Agility() :  Natural;
  playObjs() :  List(OBJECT-ID);
  playRoom() :  ROOM-ID;
  release-all(playObjs)
    post playObjs() := nil
  enter2RmI(r) post playRoom := r
  loseSt() post (Strength():= Dfstg)
    and (Agility():= Dfagl)
events
   (Life-Pt() = 0)
end interface
```

```
coordination interface Objsplay2Release-CI
partner type OBJECT
Datatypes
  PLAYER-ID, OBJECT-ID;
services
  ObjId():  OBJECT-ID;
  ObjAssign() :  PLAYER-ID;
  released() post ObjAssign() := nil
end interface
```

Apart from that specificities, all other features remain the same as in neighboring movement leading to the following coordination law.

```
coordination law Move2RmI-CL
partners
  play:PlayerMv2RmI-CI;
  Objsplay[|play.playObjs|]:Objsplay2Release-CI
rule Mvt-dead.player
when (play.Life-Pt()=0)
  with
   for i:= 1 to |play.playObjs|
   (Objsplay[i].ObjId() ∈ playObjs) and
  (Objsplay[i].ObjAssign() = play.playId())
  do play.release-all()
  for i:= 1 to |play.playObjs|
  (Objsplay[i].ObjAssign()= nil) and
  play.enter2RmI(r)
  play.loseSt()
end law
```



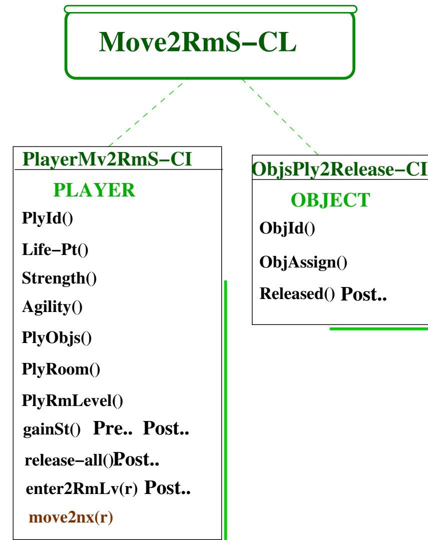
A.2.2 Next Level Player Move

A player jumping to a next level gains state points (i.e. agility and strength). Besides that, the level identity is required so that it can be changed in the law. The moving player has also to release all objects before jumping to the starting room in the next level, and may increase his state by gaining some points in life,agility and/or strength (we use for that the service `gainSt()`).

```
coordination interface playerMv2RmLv-CI
partner type PLAYER
Datatypes
  PLAYER-ID, OBJECT-ID, ROOM-ID, LEVEL-ID;
services
  playId():  PLAYER-ID;
  Life-Pt() :  Natural;
  Strength() :  Natural;
  Agility() :  Natural;
  playObjs() :  List(OBJECT-ID);
  playRoom() :  ROOM-ID;
  playRmLevel() :  LEVEL-ID;
  release-all(playObjs)
    post playObjs() := nil
  enter2RmLv(r)
    post playRoom := r and playRmLevel():=+1
  gainSt()
    pre (Let s > Strength() and a > Agility()
          and p > Life-Pt())
    post (Strength():= s) and Agility():= a)
          and Life-Pt():= p)
events   move2nx(r)
end interface
```

```
coordination interface Objsplay2Release-CI
partner type OBJECT
Datatypes
  PLAYER-ID, OBJECT-ID;
services
  ObjId():  OBJECT-ID;
  ObjAssign() :  PLAYER-ID;
  released() post ObjAssign() := nil
end interface
```

```
coordination law Move2RmS-CL
partners
  play:PlayerMv2RmS-CI;
  Objsplay[|play.playObjs|]:Objsplay2Release-CI
rule Mvt-NxtLevel.player
when (play.move2nx())
  with
   for i:= 1 to |play.playObjs|
   (Objsplay[i].ObjId() ∈ playObjs) and
   (Objsplay[i].ObjAssign() = play.playId())
  do play.release-all()
  for i:= 1 to |play.playObjs|
   (Objsplay[i].ObjAssign()= nil) and
  play.enter2RmLv(r)
  play.gainSt()
end law
```



A.3 Players' fighting

The fighting is a complex action composed of several sub-actions. First, one of the player invites another for a fight. Once this invitation is accepted by the second player (i.e. defender), the first player (i.e. the attacker) constructs his strategy by choosing two objects in his possession, and communicates one of them to the defender. The fight properly saying is composed of two rounds unless one of the two players is hit in the first round.

In the following we specify the interaction involved in each of these four sub-actions in detail.

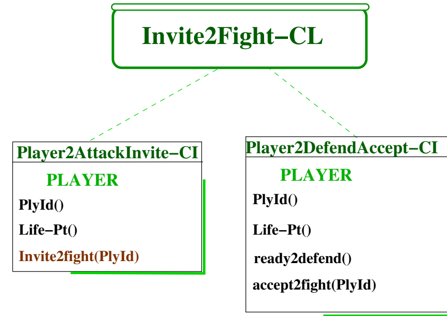
A.3.1 Invitation to Fighting

An invitation for fighting consists in sending a fighting invitation from a player (attacker) to another one (defender). When the later is ready for such a fight (e.g. not already engaged in other actions such fighting, talking with other players or has just finished a fight) the next step in the fight can be reached.

```
coordination interface Player2AttackInvite-CI
partner type PLAYER
Datatypes
  PLAYER-ID;
services
  playId(): PLAYER-ID;
  Life-Pt() : Natural;
events
  invite2fight(play:PLAYER-ID)
end interface
```

```
coordination interface Player2DefendAccept-CI
partner type PLAYER
Datatypes
  PLAYER-ID;
services
  playId(): PLAYER-ID;
  Life-Pt() : Natural;
  ready2fight() : Boolean;
  accept2fight(play:PLAYER-ID)
end interface
```

```
coordination law Invite2Fight-CL
partners play1 :  Player2AttackInvite-CI;
         play2 :  Player2DefendAccept-CI;
rule Invitation4Fight
when (play1.invite2fight(play2.playId())
     with (play2.ready2defend() = true)
       and (play*.Life-Pt()>0)
     do play2.accept2fight(play1.playId())
end law
```



`(play*.Life-Pt()>0)` is equivalent to `(play1.Life-Pt()>0 and play2.Life-Pt()>0)`, that is, both players life-points is greater than zero (they are alive).

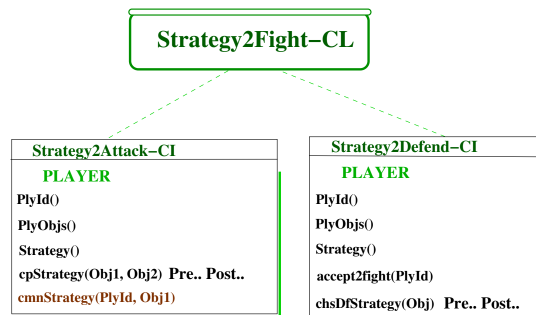
A.3.2 Choosing Fighting Strategy

After the invitation fight being accepted by the defender, the next step is for the attacker to construct his strategy by choosing two objects in his possession (using the service `CpStrategy`). One of these two selected objects has to be communicated to the defender (using the service `cmnStrategy`) so that he can choose in his turn the appropriate object to defend himself.

```
coordination interface Strategy2Attack-CI
partner type PLAYER
Datatypes
  PLAYER-ID, OBJECT-ID;
services
  playId():  PLAYER-ID;
  playObjs() :  List[OBJECT-ID];
  Strategy() :  List[OBJECT-ID];
  CpStrategy(Obj1, Obj2:  OBJECT-ID)
    pre (Obj1, Obj2 ∈ playObjs())
    post Strategy() := [Obj1.Obj2]
events
  cmnStrategy(play:PLAYER-ID,
      obj1:OBJECT-ID)
end interface
```

```
coordination interface Strategy2Defend-CI
partner type PLAYER
Datatypes
  PLAYER-ID, OBJECT-ID;
services
  playId():  PLAYER-ID;
  playObjs() :  List[OBJECT-ID];
  Strategy() :  List[OBJECT-ID];
  accept2fight(play:PLAYER-ID)
  ChsDfSrategy(ObjD:OBJECT-ID)
    pre (ObjD ∈ playObjs())
    post Strategy() := [ObjD]
end interface
```

```
coordination law Strategy2Fight-CL
partners
  play1 :  Strategy2Attack-CI;
  play2 :  Strategy2Defend-CI;
rule Strategy for Fight
when (play1.cmnStrategy(play1.playId(),
        obj1))
 with (play2.accept2fight(play1.playId()))
 do play1.cpStrategy(obj1, obj2)
  and (play2.chsDfStrategy(objD))
end law
```



A.3.3 Fighting: First Round

Once the attacker and the defender have chosen their respective strategies, a first round of fighting may take place. The states of the chosen objects for fighting as well as for defending

play a relevant role. In addition, a specific agility points may be required for using some particular objects (we denote by `rqAgility-Pt`). The final result could either be a win for the attacker (i.e. loss for the defender) or for the defender (resp. loss for attacker). That is, for sake of simplicity we do not consider the case of neutral result.

```
coordination interface Attacking1StRd-CI
partner type PLAYER
Datatypes
  PLAYER-ID, OBJECT-ID;
services
  playId(): PLAYER-ID;
  playObjs() : List[OBJECT-ID];
  Agilty-Pt() : List[OBJECT-ID];
  Strategy() : List[OBJECT-ID];
  win1st()
    post (Let p > 0, Life-Pt() := +p)
  lose1st()
    post (Let p > 0, Life-Pt() := -p)
events
  fight1stRd(play:PLAYER-ID,
    obj1:OBJECT-ID)
end interface
```

```
coordination interface Defending1StRd-CI
partner type PLAYER
Datatypes
  PLAYER-ID, OBJECT-ID;
services
  playId(): PLAYER-ID;
  playObjs() : List[OBJECT-ID];
  Agility-Pt() : List[OBJECT-ID];
  Strategy() : List[OBJECT-ID];
  defend1st()
    post (Let p > 0, Life-Pt() := +p)
  Hit1st()
    post (Let p > 0, Life-Pt() := -p)
end interface
```

```
coordination interface Obj4Fight&Defend-CI
partner type OBJECT
Datatypes
  PLAYER-ID, OBJECT-ID; OBJECT-ST
services
  ObjId(): OBJECT-ID;
  ObjState() : OBJECT-ST;
  ObjAssign() : PLAYER-ID;
  rqAgility-Pt() : natural
  Hit()
    pre (ObjAssign() = nil)
    post ObjState() := "broken"
end interface
```

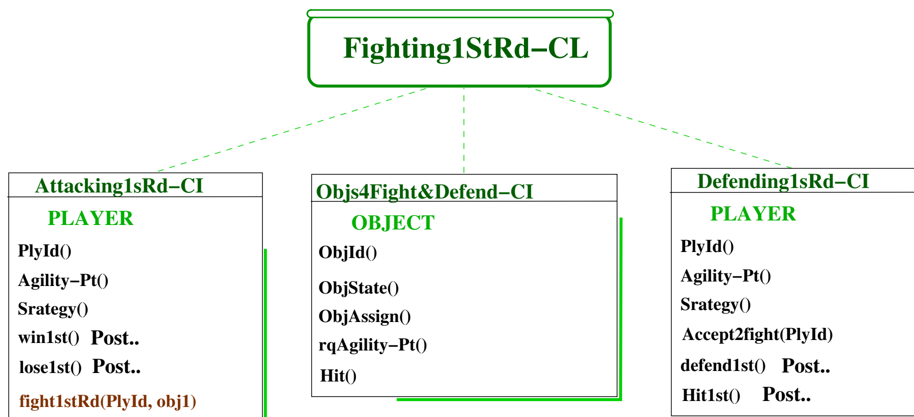


Figure 7: The Interaction Requirements for the fight first round.

```
coordination law Fighting1StRd-CL
partners play1 :  Attacking1StRd-CI;
      play2 :  Defending1StRd-CI;
      obj2play1, obj2play2 :  Objs4Attack&Defend-CI;
rule Fight1stRound
when (play1.fight1stRd(play2.playId(), obj1))
  with (play2.accept2fight(play1.playId())) and
    obj2play1.ObjAssign()=play1.playId()) and
    obj2play2.ObjAssign()=play2.playId()) and
    obj2play1.rqAgility-pt() < play1.Agility-Pt()) and
    obj2play2.rqAgility-pt() < play2.Agility-Pt()) and
    obj2play1.ObjSate() ≠ "broken" and
    obj2play1.ObjSate() ≠ "broken" and
  do
    if (obj2play1.Agility-Pt() > obj2play2.Agility-Pt())
    then play1.win1st() and
    play2.Hit1st() and obj2play2.Hit()
    if (obj2play2.Agility-Pt() > obj2play1.Agility-Pt())
    then play2.defend1st() and
    play1.lose1st() and obj2play1.Hit()
end law
```

A.3.4 Fighting: Second Round

The second round of fighting does not present any special features with respect to the first round. For that reason we skip it and let it as simple exercise for the reader, by depicting just the required services from the players and respective involved objects.

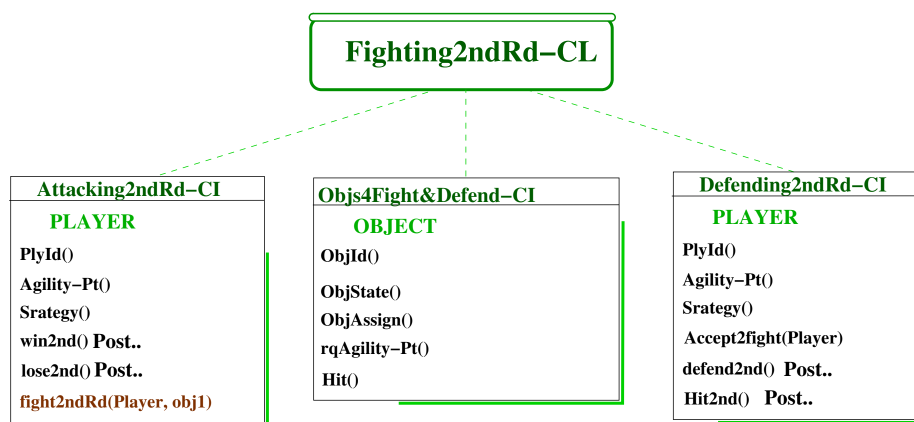


Figure 8: The Interaction Requirements for the second round.

Fighting as a whole action. We close this section by recalling the four steps required for fighting in the Figure 9 below.

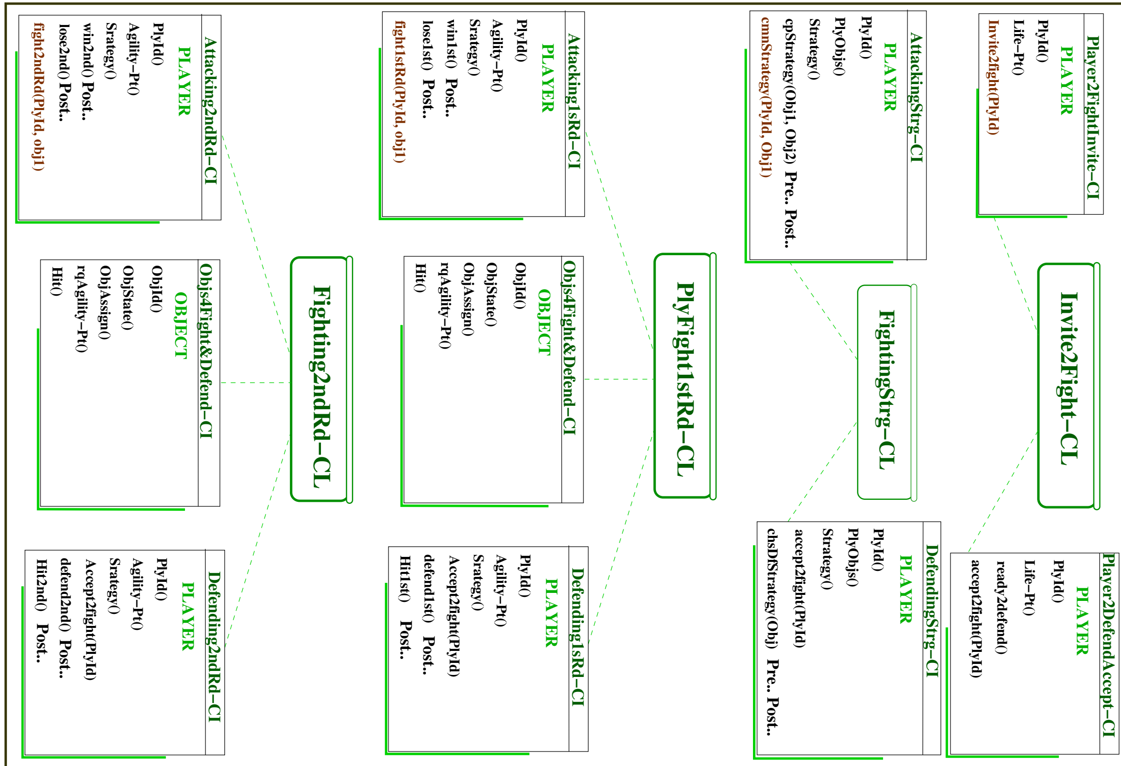


Figure 9: The four fighting steps as a package of interactions

B Location concerns : Continuation

B.1 Object trading

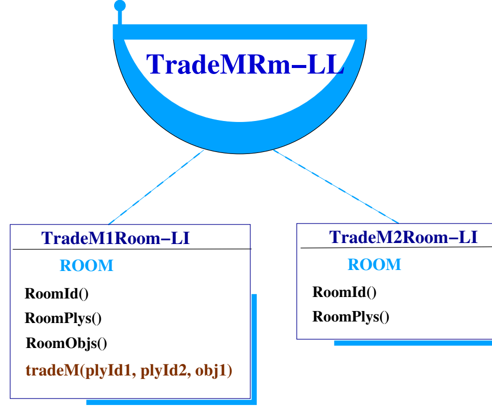
B.1.1 Trading with money

```
location interface TradeM1Room-LI
location type ROOM
datatypes
  ROOM-ID, PLAYER-ID, OBJECT-ID
services
  RoomId() :  ROOM-ID;
  Roomplays() :  List[PLAYER-ID];
  RoomObjs() :  List[OBJECT-ID];
events    tradeM(play1, play2 :  PLAYER-ID,
           obj :OBJECT-ID)
end interface
```

```
location interface TradeM2Room-LI
location type ROOM
datatypes
  ROOM-ID, PLAYER-ID
services
  RoomId() :  ROOM-ID;
  Roomplays() :  List[PLAYER-ID];
end interface
```

In the corresponding location law just one object identifier is required, which has to belong the room of the first player. We note here that like for ”message” in talking, the ”money” is not of interest to the location concerns.

```
location law TradeMRm-LL
locations rm1:  TradeM1Room-LI;
         rm2:  TradeM2Room-LI;
rule :  Talk from Room(s)
when rm1.tradeM(play1,play2, obj)
  and BT(rm1.RoomId(),rm2.RoomId())
  with (play1 ∈ rm1.Roomplays()) and
    (play2 ∈ rm2.Roomplays()) and
    (obj ∈ rm1.RoomObjs())
  do return true
when rm1.tradeM(play1,play2, obj)
  and ¬ BT(rm1.RoomId(),rm2.RoomId())
  with false
  do return false
end law
```



B.2 Player's Moving

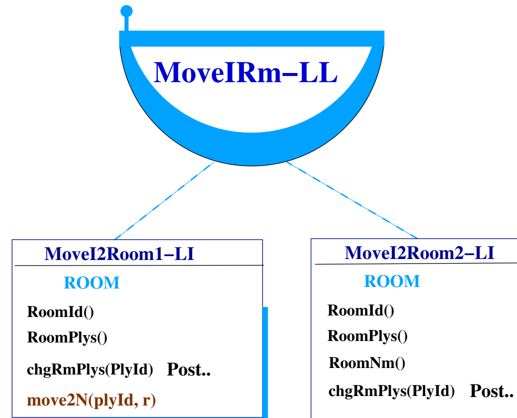
B.2.1 Moving to the Initial Room

To capture the movement of a dead player to the starting room practically all information we required for a neighboring move are required except that there is no need for doors as the jump is a forced one.

```
location interface MoveI2Room1-LI
location type ROOM
datatypes
  ROOM-ID, PLAYER-ID
services
  RoomId() :  ROOM-ID;
  Roomplays() :  List[PLAYER-ID];
  chgRmplays(play:PLAYER-ID)
    post remove(play, Roomplays())
events
  move2I(play, PLAYER-ID, r :  ROOM-ID)
end interface
```

```
location interface MoveI2Room2-LI
location type ROOM
datatypes
  ROOM-ID, PLAYER-ID, ROOM-NAME
services
  RoomId() :  ROOM-ID;
  Roomplays() :  List[PLAYER-ID];
  RoomNm() :  ROOM-NAME;
  chgRmplays(play:PLAYER-ID)
    post add(play, Roomplays())
end interface
```

```
location law MoveIRm-LL
locations rm1:  MoveI2Room1-LL;
         rm2:  MoveI2Room2-LL;
rule :  Moving to starting room
when rm1.Move2N(play :PLAYER-ID, r:ROOM-ID)
  and REACH(rm1.RoomId(),rm2.RoomId())
  with (play ∈ rm1.Roomplays()) and
    (rm2.RoomNm() = "starting")
  do rm1.chgRmplays(play) and
    rm2.chgRmplays(play)
when rm1.Move2I(play :PLAYER-ID, r:ROOM-ID)
  and ¬ REACH(rm1.RoomId(),rm2.RoomId())
  with false
  do return false
end law
```



The reachability here should exist between any room and the starting room so that any dead player can be moved in consequence.

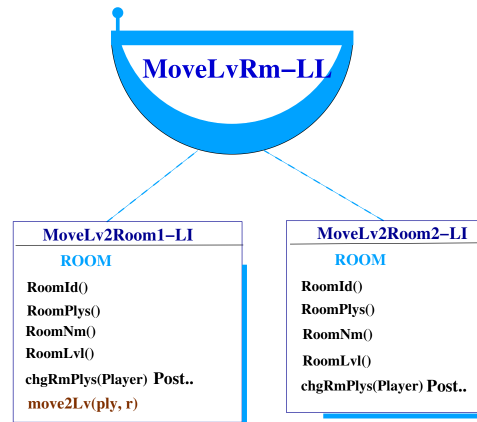
B.2.2 Moving to Next Level

To move for next level, the source room name should be "special". We also require the current *level* of that special room so that the move (to next level starting room) can be controlled.

```
location interface MoveLv2Room1-LI
location type ROOM
datatypes
  ROOM-ID, PLAYER-ID,
  ROOM-NAME,ROOM-LEVEL
services
  RoomId() :  ROOM-ID;
  Roomplays() :  List[PLAYER-ID];
  RoomNm() :  ROOM-NAME;
  RoomLvl() :  ROOM-LEVEL;
  chgRmplays(play:PLAYER-ID)
    post remove(play,Roomplays())
events
    move2Lv(play:PLAYER-ID, r :  ROOM-ID)
end interface
```

```
location interface MoveLv2Room2-LI
location type ROOM
datatypes
  ROOM-ID, PLAYER-ID,
  ROOM-NAME, ROOM-LEVEL
services
  RoomId() :  ROOM-ID;
  Roomplays() :  List[PLAYER-ID];
  RoomNm() :  ROOM-NAME;
  RoomLvl() :  ROOM-LEVEL;
  chgRmplays(play:PLAYER-ID)
    post add(play, Roomplays())
end interface
```

```
location law MoveLvRm-LL
locations rm1:  MoveLv2Room1-LL;
        rm2:  MoveLv2Room2-LL;
rule :  Next level moving
when rm1.Move2Lv(play :PLAYER-ID, r:ROOM-ID)
  and REACH(rm1.RoomId(),rm2.RoomId())
  with (play ∈ rm1.Roomplays()) and
    (rm1.RoomNm() = "special") and
    (rm2.RoomNm() = "starting") and
    (rm2.RoomLvl() = rm2.RoomLvl()+1)
  do rm1.chgRmplays(play) and
    rm2.chgRmplays(play)
when rm1.Move2Lv(play :PLAYER-ID, r:ROOM-ID)
  and ¬ REACH(rm1.RoomId(),rm2.RoomId())
  with false
  do return false
end law
```



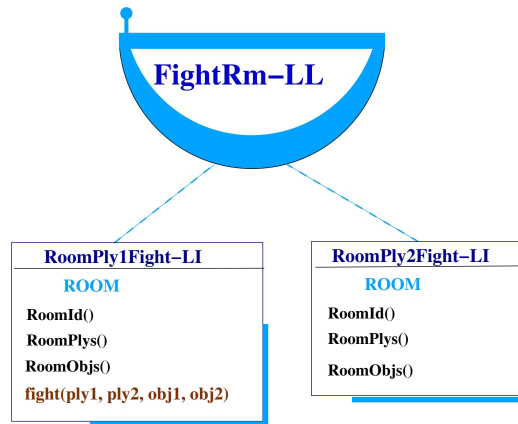
B.3 Players' fighting

¿From the location perspective, the fighting between players just imposes that they and their (used) objects must be in the same room.

```
location interface Roomplay1Fight-LI
location type ROOM
datatypes
  ROOM-ID, PLAYER-ID, OBJECT-ID
services
  RoomId() :  ROOM-ID;
  Roomplays() :  List[PLAYER-ID];
  RoomObjs() :  List[OBJECT-ID];
events    fight(play1, play2:PLAYER-ID,
           obj1, obj2:OBJECT-ID)
end interface
```

```
location interface Roomplay2Fight-LI
location type ROOM
datatypes
  ROOM-ID, PLAYER-ID, OBJECT-ID
services
  RoomId() :  ROOM-ID;
  Roomplays() :  List[PLAYER-ID];
  RoomObjs() :  List[OBJECT-ID];
end interface
```

```
location law FightRm-LL
locations rm1:  Roomplay1Fight-LI;
         rm2:  Roomplay1Fight-LI;
rule :  Fighting
when rm1.fight(play1,play2, obj1,obj2)
  and BT(rm1.RoomId(),rm2.RoomId())
  with (play1 ∈ rm1.Roomplays()) and
    (play2 ∈ rm2.Roomplays()) and
    (obj1 ∈ rm1.RoomObjs()) and
    (obj2 ∈ rm2.RoomObjs()) and
  do return true
when rm1.fight(play1,play2, obj1,obj2)
  and ¬ BT(rm1.RoomId(),rm2.RoomId())
  with true
  do return false
end law
```



C Integration of Concerns : Continuation

C.1 The players fighting action at rooms

As we have developed, the fighting from the coordination perspective is a complete package of laws that have to be superposed and performed in a sequence but as an atomic action. To reflect the exact fight rules of the game, this package has to be synchronized the corresponding single location law. The integration of these two concerns for fighting is illustrated through the Figure 10

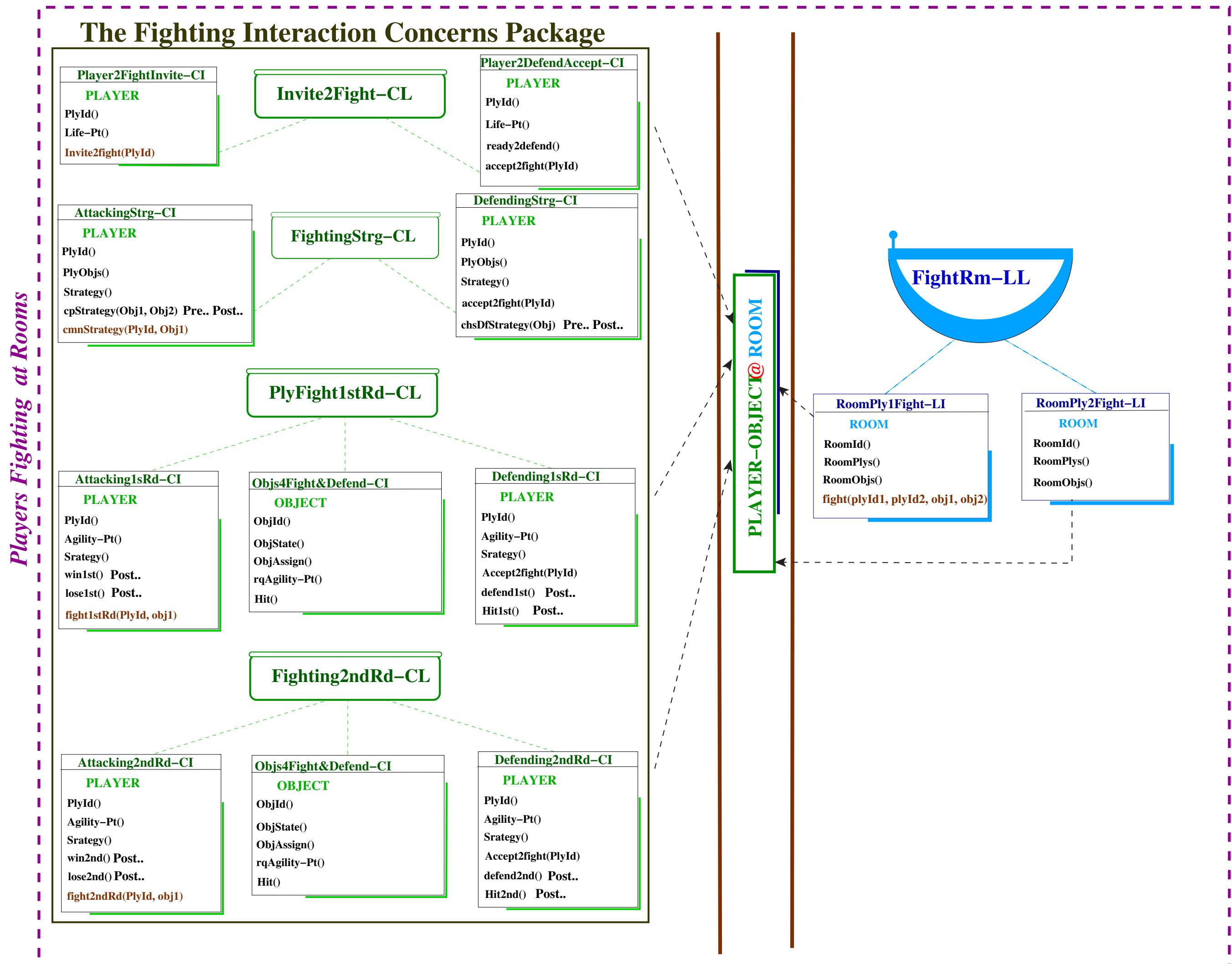


Figure 10: The C/L-based fighting action at rooms