Towards a Game-based Architecture for developing Complex Interactive Components in Computational Logic

Kostas Stathis
Intelligent Communication Systems
Department of Electrical & Electronic Engineering
Imperial College of Science, Technology and Medicine
Exhibition Road, London SW7 2BT, UK.
Email: kostas.stathis@ic.ac.uk.

Abstract

We present a game-based architecture for developing complex interactive components in computational logic. This supports the component oriented development of interactive systems, in such a way, that interactive components can be represented either as players that make moves according to the rules of a game or as umpires that enforce the rules on players, and thereby controlling their interactions. The architecture has already been successfully applied to develop knowledge-based front-ends to software packages, including multi-lingual support for such applications. The potential of the architecture is that centralized organizations controlled by umpires or decentralized organizations in terms of autonomous players can be combined to form interactive systems of a very complex nature.

1. Introduction

The emergence of the network as a communication medium has changed our conception of what a computer system is by suggesting that "the network is the computer". This change of perspective has also changed the way software is constructed, the major characteristic of this ongoing paradigm shift is that software development is now best understood as a way of systematically combining and/or aggregating components, possibly over the network. Components can be either existing – in which case the software under development reuses functionality already available, or built from scratch – in which case the software under development is augmented with new functionality.

A lot of work, both at a theoretical and practical level, is attempting to make the component-oriented development of complex software systematic. Most of the attempts end up asking similar but equally important questions, namely: what is a component, how is a component specified, how is a component implemented, how functionality of a component is expanded or filtered, how components are aggregated and co-ordinated if they interact with each other?

In this paper we present an architecture for developing complex interactive systems from simple components by using an existing logic programming framework, presented in [9-13], based on games. This framework was introduced to address inadequacies of existing logic programming approaches to develop front-ends, specifically the application of the QtU model [6] to support interactions in the front-end GLIMPSE [14], and the application of the transitions rules model [4] to support interactions in the front-end FAST [7].

Motivated by the construction of front-ends, the basic idea of games - as presented in [9] - is that the rules governing the interactions of an interactive system are seen as the rules specifying a game. Under this metaphor interactions made by the participants of an interactive system are interpreted as the moves selected by the players of a game. Specifying the valid moves of a game corresponds to defining the preconditions of actions in the interactive system. The effects of a move on the state of the game correspond to the effects of an action on the state of the interactive system. Games with more than one player correspond to interactive systems with more than one interactive participant. To implement an interactive system we construct an umpire. This is a component that displays the current state of the game in some appropriate fashion, provides means by which users can select their moves, enforces compliance of the players with the rules of the game and thereby controls the interaction. An important characteristic of the umpire is that it allows variations of play. For example, through an umpire, players may play the game from physically distant locations over a network as in networked-games. This resembles interactive systems whose participants interact from physically distant places [3].

The main advantage of thinking in these terms is that specifying an atomic game and implementing it by means of an umpire is easy. Complex interactive systems can then be built up as "compound games" composed from simpler sub-games. As an example of a compound game consider a "master's game" of chess, where a chess master plays several chess games with different student players simultaneously. Of course, as illustrated in [9], in an interactive system the component sub-games are not normally all instances of a single game as in master's chess; they would normally correspond to different application programs [3], but the system as whole may be viewed as a compound game nevertheless.

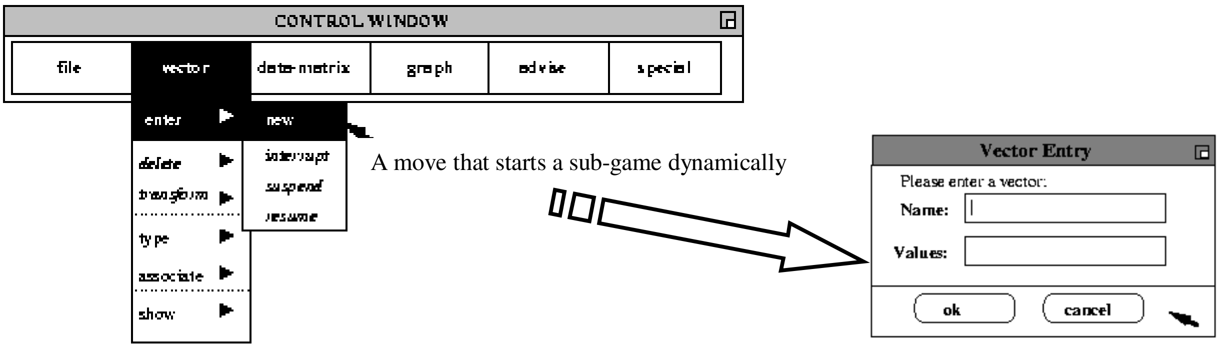

Figure 1: Interacting with the Control Panel of LAST

To reconstruct our work with GLIMPSE and FAST we applied games to build LAST [11], a system exemplifying how to use the metaphor to develop a front-end to the statistical package GLIM. Figure 1 shows how interactions in LAST are organised as compound games and presented in the form of a control panel. The figure shows a user trying to use the front-end to enter a new statistical vector. Different items in the panel are viewed as (sub) games (e.g. "vector"), which in turn can be compound games containing active (sub) games (e.g. "enter") from which the user selects a move (e.g. "new"). A move may start a sub-game dynamically (e.g. the "Vector Entry" window). The benefit of this approach is that even the addition of advice-giving components to the interactive system [8,14] can be done in a structured and natural way, as if we were providing an advisor for the corresponding game. Both context sensitive and strategic advice in LAST has been formulated in terms of best moves as determined by some expert strategy. In fact more than one advisor can be present in a game in the same way that different types of advice may be incorporated in an interactive system.

Another strength of the game metaphor is that it supports a spectrum of different organisations for an interactive system. At one end of the spectrum we have centralised organisations controlled by an umpire component, and where players are components that are normally passive unless explicitly activated by the umpire to make moves. This end has been fully explored in the application of games for building front-end applications of the type exemplified by LAST. At the other end of the spectrum, we have autonomous players that play the game by sharing a common version of the rules and without the presence of an umpire. This paper can be seen as a preliminary study of how to adapt the existing framework of umpired play to one where play is achieved via autonomous players, with the aim of combining umpires and players to form more complex interactive components.

The paper is organised as follows. Sections 2 and 3 present the existing logic programming representation for atomic and compound games respectively. In this framework, an umpire is a component that is identified by the game it is umpiring, and where players are identified by object level programs defining how players select moves. Then in section 4 we propose a general representation for the rules of a game, further treating the state of the game, the umpire and the players as separate logic programming components. We conclude the discussion in section 5 where we sketch the structure of an autonomous player and discuss some of the issues involved in building interactive systems from autonomous components.

2. Atomic Games

In this section we describe the most important parts of an existing knowledge representation framework [11] for specifying interactive components as games. This includes the specification of the rules of a game and the implementation of a game via an umpire.

2.1 Specifying the Rules of a Game

We use the following logic program to describe the rules of a game:

```
game(State, Result):-
        terminating(State, Result).
game(State, Result):-
        not terminating(State, _),
        valid(State, Move),
        effects(State, Move),
        game(State, Result).
```

The two clauses above assume that the *State* of the game is represented as a complex Prolog term identified by a unique name identifier. The two clauses are then interpreted as follows. If a terminating state is reached, the rules of the game return *Result* as output. Otherwise, in a non-terminating state the effects of a valid *Move* are applied to produce a new state. This is obtained by implicitly updating the properties indexed by the state identifier. Interaction as a game continues in the new, implicitly updated state until termination is reached. Then to specify a specific game, we need to decide on the structure of the state term and the representation of the moves, which we represent as terms of the form:

select(Player, Move).

To complete the specification of a game we further need to specify the lower level predicates: *terminating/2*, *valid/2* and *effects/2* in the form of object level rules that access, check and manipulate a state parameter.

2.2 The structure of valid moves

In practice, for example in the development of LAST, we often want to structure the definition of valid moves further. It is convenient to express this structure in terms of *available* and *legal* moves:

```
valid(State, Move):-
        available(State, Move),
        legal(State, Move).
```

Conceptually, we may think of available moves as the moves that can be physically selected at the state of a game. Some of the available moves, however, will not always be legal. To ensure that legal moves are always selected we specify them separately. We can think of these same concepts in programming terms too: *available/2* definitions act as generators while *legal/2* definitions act as filters.

Legal moves can be structured further. In LAST, for example, there are certain types of moves that must be forced at a specific stage of the game, irrespectively of what other moves are possible. In other words, in some situations some legal moves may need to overwrite the rest of the legal moves. To express this, we define legal moves in terms of *necessary* and *possible* moves:

```
legal(State, Move):-
        necessary(State, Move).
legal(State, Move):-
        not necessary(State, _),
        possible(State, Move).
```

necessary/2 definitions always overwrite the legal moves defined by *possible/2* definitions. In this way, different applications are specified by defining the conditions under which a move is available, possible or necessary and as result implicitly derive when a move is legal and valid.

2.3 Specification of an atomic game

Consider as an example of a game specification the "Vector Entry" sub-game depicted in Figure 1 presented at the introduction. This game consists of two players, the user and the statistical package GLIM. When a user enters the name and the values of the vector at the front-end level, GLIM executes a command to record the new vector entry, and once this is done the game terminates with the front-end also recording the name of the vector. To represent the state of the "Vector Entry" game we use the term:

```
vector_entry(Id).
```

Additional state properties required to describe the game are represented as assertions indexed via the game's unique *Id*. We specify the valid moves of the game by specifying the available, necessary and possible moves. The available moves are defined as:

available(vector_entry(Id), select(P, ok(Name, Values))). *% Enter name and values for a vector.*
available(vector_entry(Id), select(P, edit(Prop, Value))). *% Edit name or values of a vector.*
available(vector_entry(Id), select(P, cancel)). *% Cancel the vector entry.*
available(vector_entry(Id), select(P, enter_vector(Name, Values, Length))) % Enter vector to back-end.

The back-end must execute the action of entering a new vector once a user has specified its name and values. As there is no other move possible to the back-end at that stage we represent it as a necessary move:

```
necessary(vector_entry(Id), select("GLIM", enter_vector(Name, Values, Length))):-
        last_move(Id, select(user, ok(Name, Values))),
        vector_length(Id, Length).
```

A vector entry is possible when the user presses "ok" for the vector to be recorded by the system:

```
possible(vector_entry(_), select(user, ok(Name, Values))):-
        string(Name),
        numbers(Values),
        not vector(Name).
```

In other words, the vector must have a name, a set of numbers and the name must be new, i.e. it should not already exist in the database of the front-end.

The effects of a move identified by the user pressing the "ok" button is represented as:

```
effects(vector_entry(Id), select(user, ok(Name, Values))):-
        length(Values, L),
        retract(last_move(Id, _)),
        assert(last_move(Id, select(user, ok(Name, Values))),
        assert(vector_length(Id, L)).
```

Finally, the terminating conditions of the game are specified as:

```
terminating(vector_entry(Id),  cancelled):-
        last_move(Id,  select(user, cancel)).
terminating(vector_entry(Id),  vector(Id,Name,Values,Length)):-
        last_move(Id, select(glim, vector_entry(Name, Values, Length))).
```

In other words the game terminates either because the user has cancelled the game or because a vector has been entered successfully. Next we discuss implementation issues, but lack of space does not permit us to give a detailed example. The reader is referred to [11] for the missing implementation details.

2.4 Implementation of an Umpire

Given a set of rules characterising a game and an initial state, execution of the *game/2* program specifies all possible plays for that game with all possible results. A "game playing" program however requires a different mode of execution, whereby players effectively select one play from all possible ones. Thus, to implement a game playing program based on the rules, it is necessary to define how players communicate their choices and, assuming a computer display, how the effects of these choices are presented on the display. We also need to specify how to remove from the display of the game when this has been terminated. To do all these, in [11] we conveniently introduced the notion of an umpire whose behaviour is defined by the Prolog program:

```
play(S, R, game(S, R)):-
      stop(S, R, terminating(S, R)).
play(S, R, game(S, R)):-
      not terminating(S, _),
      choose(S, M, valid(S, M)),
      display(S, M, effects(S, M)),
      play(S, R, game(S, R)).
```

The definition of `play/3` treats terms of a game specification as data and relies upon the use of unification to select parameters of specification terms. This technique is used for accessing parts of the specification (the rules) from the implementation (the umpire). For example, part of the term that holds the communication channel of a player can be accessed by the umpire of the game to subsequently receive that player's move as a message. In this way, the definition of `play/3` is easily portable. To port the implementation of a game to a different platform one needs to re-implement the predicates `stop/3`, `choose/3` and `display/3` in the different platform.

2.5 Checking integrity of moves

The umpire of a game must enforce the rules when different players select moves. A move is enabled in the game by developing further the `choose/3` predicate. This can be defined either by checking the move in forward mode where the umpire constructs all the valid moves from the rules and the player selects one or, in backward mode where a player selects a move and the umpire checks that it is valid using the rules. The program that follows represents checking a selected move in forward mode:

```
choose(State, select(Player, Move), Valid):-
      findall(Move, Valid, Moves),
      select(Player, State, Moves, Move).
```

Similarly, the program that follows represents checking a selected move in backward mode:

```
choose(State, select(Player, Move), Valid):-
      select(Player, State, Move),
      Valid.
```

In both definitions above, the variable *Valid* unifies at execution time with a term of the form:

valid(State, select(Player, Move))

which is the head of the rules defining when a move of a specific player is valid at a specific state. Also in both definitions above, the selection strategy for moves of a player is defined for forward mode by rules of the form:

```
select(Player, State, Moves, Move):- Conditions.
```

and for backward mode by rules of the form:

```
select(Player, State, Move):- Conditions.
```

The difference in arity between selections in the two modes above reflects their characteristic differences. In forward mode the set of moves from which a player selects from is constructed by the umpire before a selection is made, while in backward mode, a player must select a move first and the umpire checks it after this move is made. Moreover, the definition of selection rules allows us to define control strategies for players separately, as if they were defined in different components.

3. Composition of Games

It is often the case that we want to augment or restrict the functionality of an interactive system. Using games we achieve these changes by either expanding or filtering the moves of the game. Changes can be incorporated by either changing the definition of the rules or by treating the changes as a separate component. In this section we show how by structuring definition of valid moves we can obtain expansions and filters. We also discuss how to specify compound games, either as a result of changes made to a system or as a result of the system being designed in terms of components to form a complex whole.

3.1 Expansions

We have used the above structure of valid moves to define expansions for moves of a game. Expansions are important because it is often computationally desirable to allow extra-legal moves to happen in the game. For instance, many chess playing programs allow a player to move pawns backward (as when chess is played without a computer) and complain only when the move is illegal. This solution resembles real life play and it is easier to implement. If only legal moves were to be allowed, one would have to disable moves at the device level (e.g. the keyboard). Expansions avoid this situation by introducing new games whose valid moves are augmenting the definition of valid moves suggested above with the rule:

valid(Expansion, Move):-
    expands(Expansion, Original),
    valid(Original, Move).

In this way valid moves from the original game persist also in the specification of the new expanded game. To complete the definition of the expansion we also need to define the terminating conditions of the new game as well as the effects of moves which they should now include the rule:

effects(Expansion, Move):-
    expands(Expansion, Original),
    effects(Original, Move).

In other words, we also need to reuse the effects of the original game whose state is included in the state of the expanded game.

3.2 Filters

To get the opposite effect of expansions we use filters. In terms of interactive components, the development of a filtered game is analogous to situations where we have to constrain the functionality of a component, as if the component was customised to supported only a subset of the interactions that it can support. The structure of a valid move for a filtered game is defined as before but we now need the rule:

available(Filtered, Move):-
filters(Filtered, Original),
valid(Original, Move).

In other words, we generate the available moves by reusing the valid moves of the original game but we filter them by appropriately defining the legal moves at the level of the filter. Moreover, as a filtered game is always a subset of the original game, we only need to write:

terminating(Filter, Result):-
filters(Filter, Original),
terminating(Original, Result).

effects(Filter, Move):-
filters(Filter, Original),
effects(Original, Move).

to reuse directly the terminating conditions and the effects of the original state.

3.3 Compound Games

Compound games are complex games composed from simpler, possibly atomic sub-games. As an example of a compound game we have mentioned in the introduction the *master's game* of chess, where a chess master plays several chess games with different student players simultaneously. Also in the introduction we have illustrated how the idea of compound games have been applied to design the control panel of LAST. We show next how we have formulated compound games.

3.3.1 Specification issues

The main issue in specifying a compound game is how to specify the co-ordination of interactions between component sub-games. We deal with this issue by defining at the compound game level under which conditions a sub-game is *active*. We then augment the definition of valid moves presented in section 2.2 to include the rule:

valid(State, inside(SubState, Move)):-
active(State, SubState),
valid(SubState, Move).

That is, the valid moves of a compound game are defined to include the valid moves exclusive to the compound game, including the valid moves in the active component sub-games. The term:

inside(SubState, Move)

specifies a sub-move. Then, different forms of coordination are possible and can be given by suitable definitions of what an active sub-game is. We distinguish two classes of co-ordination in terms of interleaving sub-games: “free interleaving” and “constrained interleaving”. To represent the state of a compound game in free interleaving we use the term:

State ***interleave*** *free*

while to represent the state of a compound game in constrained interleaving we use the term:

State ***interleave*** *constr.*

In both cases ***interleave*** is a Prolog operator. We can now represent co-ordination of a compound game in free interleaving as:

active(State ***interleave*** *free, SubState):-*
    running(State, SubState).

The developer has to specify under which conditions moves from the sub-games can be selected freely by players of the running sub-games. Similarly, in constrained interleaving, an active sub-game is defined as:

active(State ***interleave*** *constr, SubState):-*
    overtaking(State, SubState).
active(State ***interleave*** *constr, SubState):-*
    not overtaking(State, _),
    running(State, SubState).

This last definition of *active/2* relies on domain specific constraints defining *overtaking* sub-games. These are sub-games that, at specific stages of the interaction, have higher priority than the rest of the active games that are running. In this way, by defining active rules and their underlying structure in terms of overtaking and running sub-games we can specify the co-ordination of the interactions.

Both kinds of co-ordination use the same, general definition for the effects of sub-moves specified as:

effects(State ***interleave*** *_, inside(SubState, Move)):-*
    effects(SubState, Move),
    effects(State, inside(SubState, Move)).

In other words, the effects of the sub-move are applied in the state of the sub-game of the sub-game first. If the sub-game has terminated, we also need to remove the sub-game from the list of active sub-games. For this reason, the effects of the sub-move are applied in the state of the compound game (a sub-move is also a move in the compound game). To complete the definition of the compound game we need to specify how the compound game is terminated, this definition is game specific.

3.3.2 Implementation issues

There are two main considerations in developing an umpire for a compound game. One is how active sub-games are triggered when moves are made in them, and the other is how to refresh the state of a sub-game after a move is performed in it. The first issue is concerned with the way we specify choices in the compound game, which for any interleaving mode is defined as:

```
choose(State, inside(SubState, Move), Valid):-
      trigger(State, SubState, active(State, SubState)),
      choose(SubState, Move, valid(SubState, Move)).
```

The new implementation predicate `trigger/3` serves as an implementation counterpart of the specification predicate *active/2*. The triggering of sub-game is defined for free interleaving as:

```
trigger(State interleave free, SubS, active(State interleave free, SubS)):-
      pick(State, SubS,   running(State, SubS)).
```

A new predicate `pick/3` is defined specifically to deal with the way a running game is used at the implementation level of the umpire.

Similarly, the triggering of sub-games for constraint interleaving is defined generally as:

```
trigger(State interleave constr, SubS, active(State interleave constr, SubS)):-
      accept(State, SubS, overtaking(State, SubS)).
trigger(State interleave constr, SubS, active(State interleave constr, SubS)):-
      not overtaking(State, _),
      pick(State, SubS, running(State, SubS)).
```

Now the new implementation predicate `accept/3` is defined as an implementation counter-part for the specification predicate *overtaking/2*.

The other issue of implementing a compound game is how to display a compound move:

```
display(S interleave _, inside(SubS, M), effects(S interleave _, inside(SubS, M))):-
      display(SubS, M, effects(SubS, M)),
      display(S, inside(SubS,M), effects(S, inside(SubS, M))).
```

In other words, to display a sub-move we first display the effects of the sub-move in the state of the relevant sub-game and then display the effects of the sub-move at the compound game as a whole.

4. Games revisited

The formulation presented in the previous sections is the core of our approach to develop front-ends of the kind described in [7,14]. One assumption that we made in this work was that different umpires do not need to use the rules of the same game, mainly because this situation was never an issue for the specific class of applications that we were trying to support. An implication of this was that we never needed to name the umpire as a component explicitly, mainly because the umpire was identified by the game it was umpiring. However, in our more recent work with supporting multi-lingual front-end systems [13], different umpires are indeed required to localise the system interactions for different languages. Motivated by this more recent work, we revisit the development of the game framework to generalise it both in terms of applicability and generality of knowledge representation.

4.1 The rules of a game revisited

When we specified the rules of a game in the previous sections, the state of the game was conceptually a component managed by object level rules. In this section we treat the state of the game as a logic program, a component defined by a set of `rule/3` definitions. Rules are clauses that are interpreted by the vanilla meta-interpreter `solve/2` [1]. Under these new assumptions, to define the terminating conditions of a game we write:

```
terminating(State, Result):-
      terminatingRule(State, Result, Conditions),
      solve(State, Conditions).
```

A `terminatingRule/3` holds the conditions under which the state of the game terminates and with the result specified in the head. In this spirit, we define the rest of the programs specifying a game. Also, assuming the definition of valid moves given in 3.1, to define the valid and legal moves we define what is an available, necessary and possible move as:

```
available(State, Move):-
      availabilityRule(State, Move, Conditions),
      solve(State, Conditions).

necessary(State, Move):-
      necessityRule(State, Move, Conditions),
      solve(State, Conditions).

possible(State, Move):-
      possibilityRule(State, Move, Conditions),
      solve(State, Conditions).
```

To complete the revisited specification of a game we define the effects of a move as:

```
effects(State, Move):-
      effectsRule(State, Move, Conditions, Actions),
      solve(State, Conditions),
      transact(State, Actions).
```

The introduction of `effectsRule/4` aims to separate conceptually the difference between the conditions that need to be checked before the appropriate actions are carried out for the effects of a move to take place. To interpret the actions part of an effects rule we now need to define a transactions interpreter:

```
transact(State, []).
transact(State, [Action|Actions]):-
      transactOne(State, Action),
      transact(State, Actions).
```

In the simplest form a transaction is the addition of a non-existing or deletion of an existing ground fact:

```
transactOne(S, add(Fact)):-
      ground(Fact),
      not solve(S, Fact),
      assert( rule(S, Fact, []) ).
transactOne(S, delete(Fact)):-
      ground(Fact),
      solve(S, Fact),
      retract( rule(S, Fact,[]) ).
```

Transactions can also be complex, also specified by means of further transaction rules:

```
transactOne(S, Complex):-
      transactionRule(S, Complex, Conditions, Actions),
      solve(S, Conditions),
      transact(S, Actions).
```

The specification of transactions in this way has the advantages we normally attribute to the vanilla meta-interpreter in that transaction rules are specified separately from the way we implement the transaction interpreter.

4.2 The umpire revisited

To distinguish between the use of an umpire checking the play of a game and a player playing that game we redefine the `play/3` program representing the umpire of section 2.4 by adding an extra parameter and changing the name to `umpire/4` as follows:

```
umpire(U, S, R, game(S, R)):-
      stop(U, S, R, terminating(S, R)).
umpire(U, S, R, game(S, R)):-
      not terminating(S, _),
      check(U, M, S, valid(S, M)),
      display(U, S, M, effects(S, M)),
      umpire(U, S, R, game(S, R)).
```

This new definition of the umpire implementation is conceptually clearer from that given in section 2.4, because it emphasises the main responsibilities of the umpire for checking and displaying moves made by players. Also, an umpire is now a separate component with an internal state specified by `rule/3` definitions interpreted by the vanilla meta-interpreter `solve/2`. New types of rules are required to stop the game, check the moves and display the effects. For example, stopping a game is specified as:

```
stop(U, S, R, Terminating):-
      Terminating,
      stopRule(U, S, Conditions, Actions),
      solve(U, Conditions)
      transact(U, Actions).
```

A `stopRule/4` holds the conditions that would bring about the successful termination of the game. However, in the new formulation we need also to specify the actions that the umpire needs to record the outcome of the game in the local state. We can continue the formulation of the umpire by defining how moves are checked. In forward mode, a move made is checked as:

```
check(U, select(P, Move), S, Valid):-
      findall(Move, Valid, Moves),
      solve(P,   select(S, Moves, Move)).
```

and in backward mode as:

```
check(U, select(P, Move), S, Valid):-
      solve(P,   select(S, Move)),
      Valid.
```

Displaying a move is also defined generally as:

```
display(U, S, M, Effects):-
      Effects,
      displayRule(U, M, Conditions),
      solve(U, Conditions).
```

A `displayRule/3` holds the conditions that would bring about the effects that the move has on the display of the application.

4.3 The Umpire of a Compound game

In the revisited definition of the umpire moves from a compound game are checked as before:

```
check(U, S, M, valid(S, inside(SubS, M))):-
      trigger(U, S, SubS, active(S, SubS)),
      check(U, SubS, M, valid(SubS, M)).
```

The main difference now is the way we implement the umpire to trigger a sub-game, in other words, the way we define how an umpire accepts an overtaking sub-game or how the umpire picks a running one. The programs corresponding to these definitions are given below:

```
accept(U, S, SubS, Overtaking):-
      Overtaking,
      acceptRule(U, in(S, SubS), Conditions),
      solve(U, Conditions).

pick(U, S, SubS, Running):-
      Running,
      pickRule(U, in(S, SubS), Conditions),
      solve(U, Conditions).
```

As before, the above programs rely on the introduction of the new rule types: `acceptRule/3` and `pickRule/3`. To complete the definition of a compound umpire we need also to specify how to display the effects of a compound move, the definition is here similar to the one presented in 3.3.2, so we omit it for simplicity.

5. Concluding Remarks

We have presented a game-based architecture for developing complex interactive components in computational logic. This supports the development of interactive components, in such a way, that a component is organized either as a player that makes moves according to well specified rules or as an umpire that enforces the rules on players, and thereby controlling their interactions. The paper has explored in detail one extreme of the games metaphor where a game is always played in the presence of an umpire. Such a component treats players as passive information sources activated whenever they have to select a move. An alternative view of using a game specification is to play the game from the point of view of an autonomous player and possibly without the presence of an umpire. This view is motivated by our recent work in connected community systems [5]. These are systems that support communities of users to communicate by using software agents as surrogate intelligent entities. Motivated by this class of applications we may view a user agent as a player that plays a compound game whose state represents the state of a complex community environment. If we assume a logic-programming platform with a *multi-threaded* capability [2], a suitable representation of an autonomous player is as a *thread* running the program:

```
play(P, S, R, game(S, R)):-
      exit(P, S, R, terminating(S, R)).
play(P, S, R, game(S, R)):-
      quit(P, S, R, not terminating(S, R)).
play(P, S, R, game(S, R)):-
      not terminating(S, _),
      choose(P, S, M, valid(S, M)),
      execute(P, S, M, effects(S, M)),
      play(P, S, R, game(S, R)).
```

A player concludes a game by either exiting a terminating game or by quitting a non-terminating one; quitting a game is a way of justifying autonomy in the construction of the player. Otherwise, the player plays the game by choosing a move, executing it and continuing the play until the game ends. To define a player we now need to define the predicates `exit/4`, `quit/4`, `choose/4` and `execute/4` for atomic and compound games.

What is the advantage of having autonomous players? The most important one is that we can support decentralized organizations of interactive systems and develop them in a component-oriented fashion as we have done with centralized ones. Moreover, decentralized organizations may be combined with centralized ones to form more complex systems. For example, we can build a component that most of the time acts as an autonomous player, and only in some cases – for instance when it has to interact with a user – acts as an umpire. We are currently planning to experiment with these ideas in our work with a multi-agent system that we develop in [5]. Most of the issues that arise in the autonomous case are about the implementation of synchronizing different players to access or change the state of the game now managed by an *environment process*. The detailed consideration of these issues, however, is the subject of further work.

References

[1] A. Brogi, Program Construction in Computational Logic, Department of Computing, University of Pisa, Italy, 1993.

[2] D. Chu. IC-Prolog II: *A Multi-threaded Prolog System*. In G. Succi and G. Colla (Eds), Proceedings of the ICLP-93 Post-conference Workshop on Concurrent, Distributed and Parallel Implementations of Logic Programming, 1993.

[3] E. A. Edmonds and B. S. Murray and J. Ghazikhanian and S. P. Heggie. *The Re-use and Integration of Existing Software: A Central Role for the Intelligent User Interface*. HCI '92, People and Computers VII, 1992, pages 415-427, British Computer Society Conference Series, A. Monk and D. Diaper and M.D. Harisson (Eds), Cambridge University Press.

[4] M. Green. *A Survey of three Dialogue Models.* ACM Transactions on Graphics, 5(3) : 244 – 275, 1986.

[5] J. Pitt, A. Mamdani and K. Stathis, *A Cooperative Information System for the Connected Communities*, Proceedings of COOP'98, vol 2: 72-76, Cannes, France, May 1998.

[6] M. J. Sergot, A Query-the-User facility of Logic Programming. In P. Degano and E. Sandwell (Eds), Integrated Interactive Computer Systems, pp 27-41. North Holland, 1983.

[7] K. Stathis, *A FAST Front-End Application*, Proceedings of the 2nd International Conference of Practical Applications of Prolog, London, April 1994.

[8] K. Stathis, *How to give FAST Advice*, Proceedings of the 7th Symposium and Exhibition on Industrial Applications of Prolog, Tokyo, Japan, October 1994.

[9] K. Stathis and M.J. Sergot, *Games as a Metaphor for Interactive Systems*, 7th International Conference of Human Computer Interaction, HCI'96, Springer-Verlag, August 1996.

[10] K. Stathis and M.J. Sergot, *A Fast Glimpse of Knowledge-Based Front-Ends as Games*, Proceedings of the 1st International Conference on Successes and Pitfalls of Real-World Applications in Knowledge-Based Systems, pages 81-90, AIT Press, Thailand, September 1996.

[11] K. Stathis, *Game-Based Development of Interactive Systems*, Ph.D. Thesis, Department of Computing, Imperial College of Science, Technology and Medicine, London, SW7 2BZ, UK, November 1996.

[12] K. Stathis and M.J. Sergot, *Knowledge-Based Front-Ends as Games*, in Journal of Lessons Learned in Information Systems Management, 2(1), pages 135-147, Jay Liebowitz Editor, 1997.

[13] K. Stathis and M.J. Sergot, *An Abstract Framework for Globalising Interactive Systems*, Journal of Interacting with Computers, 9(4), pages 401-416, D. Day Guest Editor, 1998.

[14] D.E. Wolstenholme, C.M. O'Brien and J.A.Nelder. *GLIMPSE: a Knowledge-Based Front-End for statistical analysis*. Knowledge-Based Systems, 1:173-178, 1988.