Player Collaboration in Virtual Environments using Hierarchical Task Network Planning*

Daniele Masato, Stuart Chalmers and Alun Preece
University of Aberdeen, Computing Science, Aberdeen, UK
{dmasato,schalmer,apreece}@csd.abdn.ac.uk

Abstract

In recent years, the fast evolution in computer games has moved government organizations to investigate how they can be exploited as virtual environments to simulate scenarios which could be expensive or even dangerous to set up in real life, in particular when collaboration among humans is required in order to carry out a shared plan. The proposed system allows the tracking of progresses achieved by each plan participant within the planning domain, by mapping its steps to states and humans' actions in the virtual environment. It also permits to render the environment and the planning software loosely coupled and to provide flexible responses to participants' actions in the form of different alternatives to the same plan, such that goals can be achieved following different courses of action.

1 Introduction

In recent years computer games have evolved rapidly both graphically and from a playability point of view, exploiting the more and more powerful hardware resources and becoming increasingly immersive, realistic and compelling. This evolution has moved several government organizations to launch various initiatives about *Serious Gaming*[1], that is the study of how computer games can be used as a virtual environment to simulate scenarios which could be expensive, difficult or even dangerous to set up in real life. These scenarios range from military applications such as house search and clearance or foot patrols to industrial applications like safety analysis and accident prevention, where they are employed as a teaching support to train the target audience. This approach could also be used in *Noncombatant Evacuation Operations* and *Peace Keeping Operations* simulations [1]: the former in which military forces are needed to evacuate people whose lives are in danger, trying at the same time to minimize the risk of combat zones [4] while the latter consists of activities that impartially makes use of diplomatic, civil and military means to restore or maintain peace [3]. Moreover, virtual environments are suitable to study individual and

*This research is continuing through participation in the International Technology Alliance sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence. See http://www.usukita.org/

[1] Several government initiatives about Serious Gaming can be found in these websites: http://www.seriousgames.org and http://www.defencegaming.com.

collaborative behaviours emerging from teams consisting of players with different cultural/social backgrounds carrying out the same tasks. Examples of using a virtual environment in training and research activities include DIVE2[2], SABRE[3] [9] and Virthualis[4].

This paper focuses on interactions among human players (and possibly software agents) in a virtual environment where collaboration is required in order to carry out a shared plan. Since a plan can be hierarchically decomposed in a series of smaller tasks assigned to each participant involved in the operations, the main aim consists in tracking the progress achieved by the players within the planning domain, by mapping its various steps to states and players' actions in the virtual environment.

2 The Virtual Environment: Battlefield 2

The virtual environment chosen for our work is *Battlefield 2*[5], a first-person shooter game with strategy elements, in which players fight in a realistic battlefield. The publisher[6] provides a complete scenario editor to build custom modifications and scenarios for the game. Furthermore, the game provides both a server and a client implementation, hence while players are playing in the environment rendered by their clients, the server manages all the game logic and represents the right place to gather data about events happening in the scenario. The server is instrumented by means of Python scripts which can capture many types of game events, such as player spawns and deaths, vehicle use or area accessing/leaving[7]. In this way, it is possible to write a series of Python functions which will be called automatically by the game engine whenever one of the supported events happens during the match. The list of catchable events is quite extensive[8]:

Player events: triggered when a player interacts with other players/weapons and on changes to players' health;

Vehicle events: triggered when a player interacts with a vehicle. Vehicles differ from weapons because they can experience damage *and* be restored to full functionality, whereas weapons cannot;

Sensitive areas events: sensitive spherical areas within the environment, generating events upon player/vehicles entry/exit;

Timer events: set up during the match, they capture elapsed time.

[2] Dismounted Infantry Virtual Environment, see http://www.defenseindustrydaily.com/2005/11/combat-sims-half-life-goes-to-afghanistan/index.php

[3] Situation Authorable Behavior Research Environment

[4] Virtual Reality and Human Factors Applications for Improving Safety, see http://www.virthualis.org/index.php

[5] See http://www.ea.com/official/battlefield/battlefield2/us

[6] Electronic Arts Ltd.

[7] Although official documentation is poor under this aspect, information can be found at http://bf2tech.org and http://bfeditor.org.

[8] Notifications are also triggered when the game changes status (loading/playing/end). These events can be used to perform initialization and cleanup tasks relating to the planner.

2.1 JSHOP2 and Hierarchical Task Network planning

In *Hierarchical Task Networks (HTN)* planning, a plan is defined by a high-level abstract description of required tasks (e.g. build a house) [7]. The plan is then refined by applying a recursive decomposition process, where each task is reduced to a partially ordered set of smaller tasks (e.g. obtain a permit, dig the ground, lay foundations etc.). The process terminates when it reaches *primitive tasks*, represented by *planning operators* defined by a domain description, also called the *planning domain*. The planner needs to be instructed on how to decompose complex tasks to simpler subtasks and this is achieved by means of schemata called *planning methods*. For the same high-level task there could be a number of applicable methods, so the planner may have to perform several attempts before finding a suitable decomposition to a lower level.

We have utilised *JSHOP2*[9], a planning algorithm based on *Ordered Task Decomposition*, a modified version of HTN planning involving planning for tasks in the same order that they will later be executed[10]. In this way, much of the uncertainty regarding the world represented by the planning domain is removed, and this makes it possible to know the world status at each step of the decomposition process. Moreover, additional expressive power may be added through *axioms*[11], *symbolic* and *numeric conditions* and *external function calls* which allow the planner to reason about the current world state and make decisions on this basis. Methods, operators and axioms all involve logical expressions, that are combinations of atom terms using logical operators (`and`, `or`, `not`), implications (`imply`) and universal quantificators (`forall`).

JSHOP2 does not interpret the plan dynamically but *compiles* the plan, meaning that given the planning problem and its domain, JSHOP2 transforms these into a set of Java classes (contained in the *domain-specific planner* and in the *problem*) tailored and optimized to solve the specific problem (Figure 1). This approach does not pose any difficulties, as long as the planner, which relies on a closed-world assumption, does not need to interact with an external environment. Instead, should an external event (e.g. a function call) be used to trigger a modification in the planner world state, problems would arise. In fact, the compilation process implies that all the entities defined in the domain description, such as operators, methods and axioms will be mapped to an internal, domain-specific representation (i.e. specific Java classes). Accordingly, these entities are identified by means of unique integer values (not known prior to compilation). This means that all state atoms included in the domain description will be mapped to integer values, and these will be the only references an external component may use to assert or retract such atoms in order to change the world state. In conclusion, *all the state atoms which require to be later accessible by an external software must be declared somewhere in the domain definition*, to retrieve their correspondent integer representations.

[9]See `http://www.cs.umd.edu/projects/shop`

[10]O-Plan [8] has also been considered and tested, and we believe that substituting this an alternative planner is possible due to the open architecture implemented.

[11]Axioms are simply evaluations performed on the current world state.

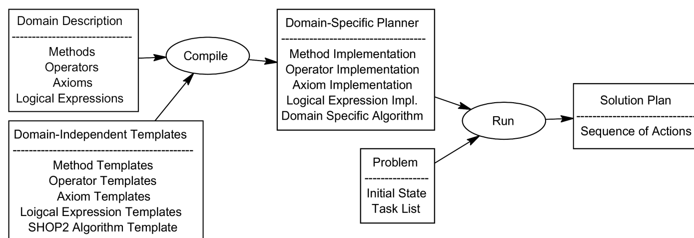

Figure 1: JSHOP2's plan compilation process (from [2], page 14)

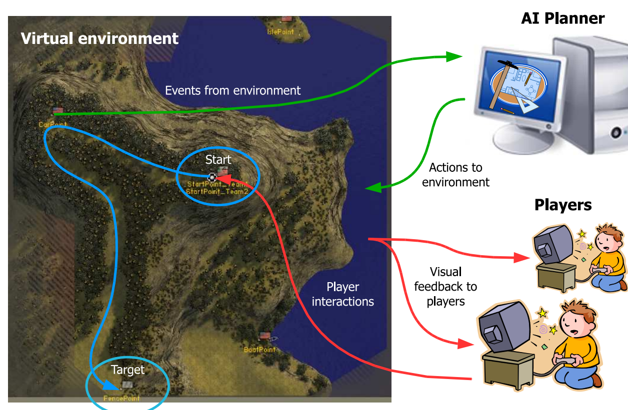

Figure 2: System overview

3 System Architecture & Design

In Figure 2 we can see the two main components of the system: the virtual environment and the planner. In this example the players are at the *Start* point on the hilltop and the plan requires that they reach the *Target* point (a fence) at the hill foot following the blue path. The players interact with the virtual environment, hosted in a server machine, using their client machines and receive visual feedback of their interactions (moving, picking up objects, driving vehicles, firing weapons).

The virtual environment maps these operations into a machine understandable format, triggering ***events*** which are sent to the planner. Since the planner has been instructed about the goal (reach the target) and how to achieve it, one or more events are applied to the world state, thus modifying and driving the plan execution. When the planner realizes that a primitive operator is applicable to the current world state, it fires one or more ***actions*** back to the environment, which transforms these into advice or directions for the players. Note that the players are not forced to follow the instructions issued by the planner.

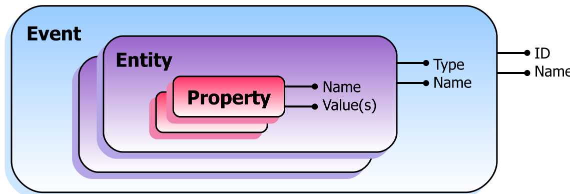

Figure 3: The event model

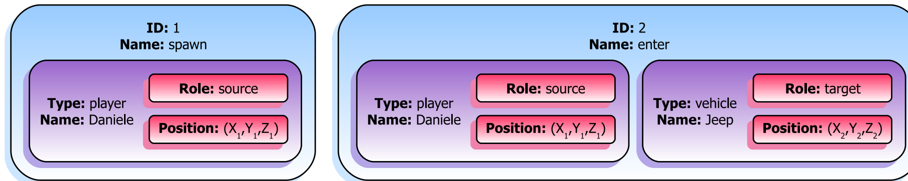

Figure 4: Two examples of events

3.1 Message modelling: Events and Actions

Before outlining the system design, we define the messages exchanged between the environment and the planner, in the form of events and actions.

Events An event is defined as something happening in the environment, which in turn causes a change in the planner world state (Figure 3). In this model, an event instance is identified by a unique integer *ID* and a *Name* indicating the type of event (e.g. spawn of player, death of player). The ID distinguishes different events as well as events with the same features and content but triggered at different times: in this way, when they are mapped into the planner world state, they maintain their own identity, allowing reasoning on the basis of *sequences of events* instead of every single event. Each event involves one or more *entities*, defined by a *Type* (e.g. players, vehicles) and a *Name* (e.g. Player1, Jeep). Each entity has one or more *properties* (e.g. position, rotation) related to the event it is involved in.

Two instances of event are shown in Figure 4. The first shows the player entity *Daniele* has just spawned in the environment at coordinates (X_1, Y_1, Z_1). The second shows that the player has entered the vehicle entity identified by *Jeep*, located in position (X_2, Y_2, Z_2)[12].

Actions An action is a primitive operator applied by the planner to its world state, causing modification or visual feedback in the environment (Figure 5). The action is identified by a *Name* (e.g. say-to-player, move-object) which indicates the environment modification caused. An action is addressed to a particular *Target*, that is an entity as above, but with no specified properties. *Properties* are moved outside the target because they are related to the action

[12] Here the property *role* states that the player is acting as the *event source*, whereas the vehicle is the *event target*, meaning that *Daniele* has entered in the *Jeep*, not the opposite.

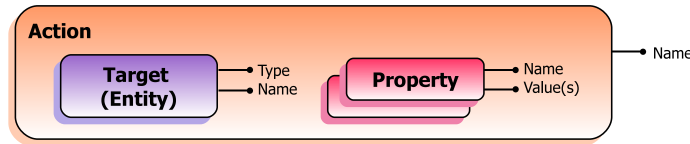

Figure 5: The action model

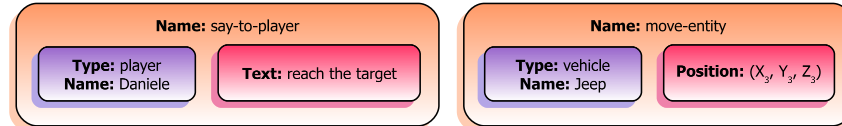

Figure 6: Two examples of action

itself and their aim is to specialize the generic modification it represents. Two instances of action are shown in Figure 6. The first is addressed to the player *Daniele* and causes a message to appear on screen. The message content is specified by the *Text* property. The second indicates that the vehicle entity *Jeep* has to be moved to the position (X_3, Y_3, Z_3).

3.2 System Architecture

This section outlines the system architecture (Figure 7), following a top-down approach, from the two web services to the calls to and from the game engine and planner. The system design is almost completely symmetric: as will be discussed later in the next sections, this design allows both *asynchronous* message exchanges between the two sides and a more rational development process. It also provides a good level of abstraction for easy extension or adaptation.

The communication channel between the environment and the planner is realized in the form of *web services*. The web services expose some methods (described in the *WSDL* format) which are called by clients. Messages exchanged between the web service and clients are enclosed in *Simple Object Access Protocol*[13] envelopes and formatted according to the XML standard.

3.3 Game side design

The Game Web Service The game web service is the recipient of actions from the planner, modelled as described in section 3.1. Actions are unwrapped from their SOAP envelope and deserialized into language-specific[14] complex data structures ready to be delivered to the level below. In order to fulfil its role, the game web service must expose at least the following methods[15]:

- `postAction`: the method the planner side will call to post its actions;

[13]See `http://www.w3.org/TR/wsdl` and `http://www.w3.org/TR/soap`
[14]Python in this case.
[15]The last two methods should cause a restart in the environment to allow another course of action for the same scenario or to load another scenario.

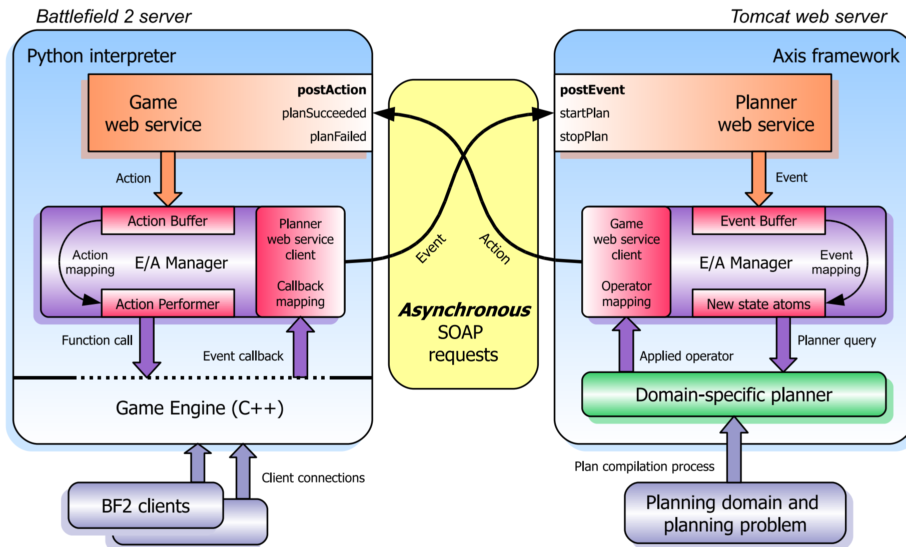

Figure 7: System architecture

- `planSucceeded`: method called when all the plan tasks have been completed successfully;
- `planFailed`: method called when all the attempts to solve the plan fail.

The Event/Action Manager This is the most important component of the system because it acts as an interpreter between the web service and the game engine, and it performs a client role with respect to the planner web service using an asynchronous communication channel[16]. In fact, the E/A Manager provides an *Action Buffer* where the game web service can enqueue received actions continuously, thus allowing the planner side to send them whenever they are available. This means that:

- the planner is not forced to post an action only in response to a received event. It may send actions related to sequences of events or actions unrelated with any event (e.g. to welcome to the game participants);
- although an event response should not arrive too late, the planner is not required to provide a solution in a fixed timeframe, increasing its ability to solve complex situations;
- the planner may map a single operator applied to its world state to more actions and post them sequentially.

When actions are available in the buffer and the environment manager is ready to process them, the *Action Performer* module dequeues the first action

[16]An asynchronous communication channel increases the decoupling of the two components, providing more flexibility as they can both proceed in parallel, helping mitigate delays in delivery or temporary bottlenecks.

and maps it to the game engine format (e.g. a function call, see later). Although the Action Performer works together with the E/A Manager, it should run in parallel as an independent unit to allow the latter to handle efficiently the callbacks triggered by the game engine and the actions enqueued in the buffer.

When an event is triggered inside the environment, the E/A Manager is notified (e.g. through a callback mechanism) and receives a raw event. The raw event must then be mapped to a complex data structure (compliant with the event model). Finally, the web service client embedded inside the E/A Manager posts the event to the planner using the `postEvent` method exposed by the planner web service.

Interfacing with the game engine The lower level consists of the interface between the E/A Manager and the environment engine. This part of the system is heavily dependent upon the engine itself and its degree of openness, therefore an abstraction of this component is not easy to devise. However, every virtual environment to be interfaced with the E/A Manager should provide these two facilities:

- a way to modify some aspects of the environment in response to actions;
- a way to notify the E/A Manager of events happening in the environment.

In Battlefield 2 server these are achieved by a set of game engine function calls, callable from the Python interpreter, which includes support for displaying messages on client screens, creating new spherical sensitive areas with a given radius and position and moving objects or vehicles in the map[17], and a callback mechanism that allows different Python functions implemented in the E/A Manager to be called by the game engine whenever players' interactions with the environment trigger some events.

3.4 Planner Side Design

The Planner Web Service The planner side design is similar to the game side, apart from the lower level where the interface to the environment is replaced with the interface to the planner. The planner web service act as a hub which collects events arriving from the planner side (and modelled as described in section 3.1), unwraps them from their SOAP envelope and finally deserializes them into language-specific complex data structures, ready to be delivered to the level below. In order to synchronize itself with the game side, the planner web service must expose at least the following methods:

- `postEvent:` the method the game side will call to post its events;
- `startPlan:` method called when the virtual environment scenario is loaded and ready to accept actions;

[17]Many other functions are available in Battlefield 2 server, but these are the most important and were used in the system implementation

- `stopPlan:` method called when the virtual environment is shut down due to an error or players' directive. This should also reset the planner to its initial state so it is ready the next time `startPlan` is called.

The Event/Action Manager This performs the same functions as its twin on the game side. In particular this E/A Manager provides an *Event Buffer* where the planner web service can enqueue received events continuously, allowing the game side to send them whenever they are produced. As a consequence:

- the environment is not required to post an event only in response to a received action. This is an extremely important consideration since the environment engine is optimized to deliver high performances during the gameplay and its overhead should be minimized[18];
- a callback in the game E/A Manager may be mapped into multiple events which the game side may send sequentially at its maximum speed.

When new events are available in the buffer and the planner needs additional information to reach a solution, it queries the E/A Manager which in turn dequeues the first event from the buffer and maps it into state atoms. These are then returned to the planner that will assert or retract them in its world state. Although many events can be triggered at the same time in the environment, they will be delivered in a particular order to the planner web server, hence the Event Buffer should be implemented as a queue to preserve such order and for consistency with its twin on the game side.

When the planner applies a primitive operator to its world state, the E/A Manager is notified and receives a raw action in a format dependent from the planner itself. The raw action must then be mapped to a complex data structure which complies with the action model for it to be sent. Finally, the web service client embedded within the E/A Manager posts the action to the environment using the `postAction` method exposed by the game web service.

Changing the planner world state This level of the architecture allows the modification of the planner world state to take advantage of the received events. Interfacing the E/A Manager with the planner is not as difficult as its counterpart on the game side, however, every planner to be interfaced with the E/A Manager should provide these two capabilities:

- a way to specify what state atoms are to be asserted or retracted in response to an event;
- an external function call support which allows to notify the E/A Manager that an operator has been applied.

JSHOP2 provide a flexible function call support through a Java interface called `Calculate`. A multi-purpose function, whose behaviour is based on the arguments it receives, has been implemented to address both the requirements.

[18]If the environment had to wait for an action in response to each event sent to the planner, this would slow down or possibly block the game engine, causing unpredictable behaviours that may range from impaired user experience to game engine crashes.

4 Implementation

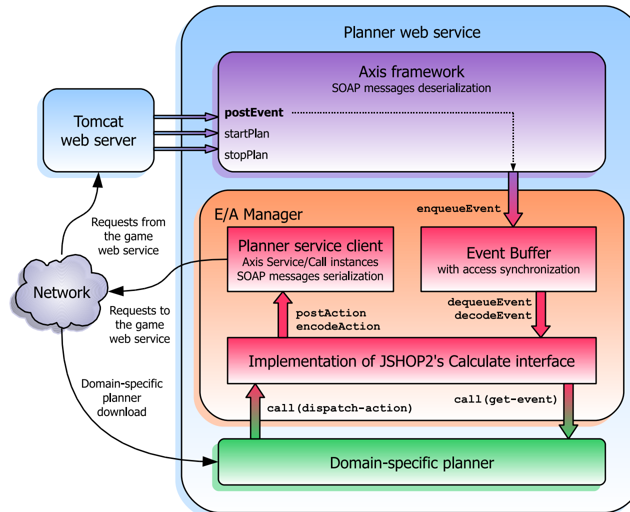

Figure 8: Planner side implementation

The system implementation follows the design in Figure 7. The virtual environment is provided by Battlefield 2: its server runs on a dedicated machine and works as a hub for all the events generated by the clients, while players connect to it from machines running the clients, which in turn render the virtual environment on players' display.

The plan is managed by JSHOP2, which generates the *domain specific planner*, allowing the description of the current plan state at each generated step [6] and permits tracing of the plan development interactively, without waiting for the planner to calculate the entire sequence of steps of the plan. JSHOP2 has already been used with good results in various kinds of similar situations [5].

Our work involves the creation of two web services, the first programmed in Python and running inside the game server, the second developed in Java (to communicate with JSHOP2) and hosted by the *Axis* framework on a *Tomcat*[19] web server. This approach has some immediate advantages: firstly, it allows Battlefield 2 and JSHOP2 to exchange messages over a network; moreover it offers a loose coupling between the virtual environment and the planner, allowing experimenting with environments other than Battlefield 2 (and the use of planners other than JSHOP2). Finally, it permits the implementation of the asynchronous communications described in section 3.

To integrate with the Python interpreter embedded in Battlefield 2 we use ZSI, the Zolera SOAP Infrastructure[20]. In particular, ZSI parses and generates SOAP messages, and converts between native Python data types and SOAP syntax, including complex data types.

[19] See `http://ws.apache.org/axis` and `http://tomcat.apache.org`
[20] See `http://pywebsvcs.sourceforge.net`

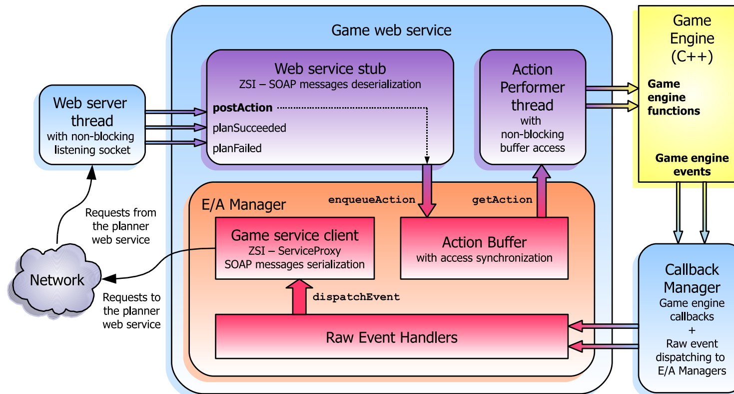

Figure 9: Game side implementation

5 Evaluation

We carried out the evaluation of our planner interaction in the system using the simple scenario detailed in section 3. The scenario requires that the player, starting from the top of a hill, completes two tasks: reach the fence at the hill foot, then pass the fence and reach the island behind the hill. To achieve these goals the player may decide whether to use the vehicles provided (a car and a boat). An excerpt from the planning domain that tracks the accomplishment of the first task is shown in Figure 10, together with the scenario definition showing the sequence of the two goals.

Currently the system tracks one player's progress in the environment. In fact, once the plan author has defined the plan and it has been compiled, the system is able to realize the mapping between the plan logic and the player's interactions with the virtual environment without any other user interventions. As previously stated, the planning problem defines the player's tasks and their execution sequence at a very high abstraction level, whereas the planning domain drives the planner toward a solution outlining how high-level tasks are decomposed in simpler tasks. The whole planning process could be seen as a rule-based system where planning operators are applied when certain conditions are met and changes in the world state allow to apply some operators rather than others.

For example, taking the `reach-fence` method, this is simply decomposed in three subtasks: notify the player of his/her task[21], wait for the task to be completed, and notify the user of the success. The same approach is applied to the subtask `wait-for-fence-reached` where the JSHOP2 constructs (such as axioms in a method precondition) make it relatively easy to follow different directions whether some conditions (the fence has been reached) are verified or not. In general, checking whether particular situations are valid in the virtual

[21] `say-to-player` is further decomposed into an operator which performs a `dispatch-action` function call to the planner E/A Manager, but the latter is not listed for space reasons.

```
(defdomain domain
...
; Given a player already in the scenario, asks the player to reach the fence
(:method (reach-fence)
   ((chosen ?player))
   (
      ; say-to-player will apply an operator to the planner world state which in
      ; turn will call the dispatch-action method in the E/A Manager
      (say-to-player ?player (Reach the fence down the hill))
      (wait-for-fence-reached ?player)
      (say-to-player ?player (You reached the fence))
   )
)
(:method (wait-for-fence-reached ?player)
   ; Waits for goal achievement, updates the world state at each loop and repeats
   ((player-available ?player) (not (fence-reached ?player)))
   (
      (try-use-land-vehicle ?player)
      ; update-state will call the get-event method in the E/A Manager
      (update-state)
      (wait-for-fence-reached ?player)
   )
   ; Goal achieved, terminates the plan
   ((fence-reached ?player))
   nil
)
; Suggests the usage of a land vehicle, if available, otherwise walking!
(:method (try-use-land-vehicle ?player)
   ; Player may drive to the fence
   ((car-available ?car) (player-in-car ?player ?car))
   ((say-to-player ?player (Drive the car to the target)))
   ; Player may reach the fence by car
   ((car-available ?car))
   ((say-to-player ?player (Use the car to reach the target faster)))
   ; Player has to reach the fence by walk
   nil
   ((say-to-player ?player (You have no car-walk to the target)))
)
; Axiom to see whether a player is available (spawned in the environment)
(:- (player-available ?player)
   ((entity (player ?idp) (event (?ide spawn)))
      (assign ?player (player ?idp)))
)
; Axiom to see whether the player has reached the fence
(:- (fence-reached ?player)
   ((entity ?player (event (?ide enter)))
      (entity (area fencepoint) (event (?ide enter))))
)
...
) ; end of domain definition

(defproblem definition domain
   ; Starts with an empty initial state
   nil
   ((wait-for-player) (reach-fence) (reach-island))
) ; end of problem definition
```

Figure 10: An excerpt of the planning domain and problem specification, represented in JSHOP2 formalism

environment is facilitated by the adoption of the event model described in section 3.1. As can be noted from the `fence-reached` axiom, it is rather straightforward to capture the situation in which the player has entered in the `fencepoint` area, because these two entities are linked by the same event and they are both part of the current world state.

In `wait-for-fence-reached` the world state update is realized by means of `update-state`[22], then the method calls itself again recursively. In this way the planner keeps checking for the `fence-reached` condition to be true, updating the world state at every loop. Events which do not contribute to satisfying this condition are asserted in the world state but are virtually filtered. In this way, events not directly related to the current task do not lead to any new action generated from the planner. Nonetheless, some of the mentioned events may be relevant as they may lead to a different alternative for the task (reaching the fence). This case is considered in the `try-use-land-vehicle` method which gives different advice to the player depending on whether a car is present and whether the player is in the car. Note that the player is not forced to follow the advice because the main task remains to reach the fence, no matter how this is accomplished.

To summarize, a carefully crafted plan definition permits various courses of action for the same plan. However, the implementation mentioned above should be use with caution, since it recursive nature might lead to stack overflow exceptions in the planner. Apart from this, the plan definition may be as complex and as detailed as resources permit, since the JSHOP2 compilation process always generates an optimized solver tailored to cope with such specific plan definition.

6 Conclusions and future work

The implementation we have is flexible enough to offer some nice additional features, such as the possibility to define alternatives for the same plan task or synchronizing multiple partially dependent plans[23]. In this case it could be possible to evaluate how players react to unexpected situations (as in emergency simulations) and to evaluate how they collaborate together, and to test the use of and ability of software agents in human/agent collaborative teams. The former feature was successfully implemented in the plan employed for evaluation purposes, whereas the latter is currently being investigated. This also touches on the scenario of plans involving multiple players. Applying a simple plan to an environment with more than one player should allow a second player to take over the current task execution should the first player fail in the attempt. For example, taking again the `reach-fence` method in Figure 10 and assuming two players have spawned in the environment, if the designated player (in `(chosen ?player)`) fails to complete the `wait-for-fence-reached` subtask, then the JSHOP2 algorithm should backtrack and bind the variable `?player` to another, attempting to complete the same subtask once more with a different player.

[22] This task is further decomposed in a `get-event` function call to the planner E/A Manager.
[23] As long as the plan author designs plans which support these aspects.

We are also investigating more interactive communication between players and the planner. Currently we use the planning framework to direct and advise the players toward the plan solution. However, if more solutions to a plan are available, it is not possible for a player to ask the planner for a specific solution, or to reject an unsatisfactory one, as the planner autonomously "decides" what is best for the player in order to achieve the goals. This extension can be implemented using the embedded communication channel among Battlefield 2 clients and the game server, which should make possible for players to issue commands from their local console to the server console. Once received by the server console, commands could be parsed and sent to the planner to replan accordingly to players' requests.

References

[1] Anders Frank. Foreign Grounds, a Digital Game for DecisionMaking in Foreign Cultures. In 2nd *Workshop on Exploring Commercial Games for Military Use*, October 2005.

[2] Okhtay Ilghami. Documentation for JSHOP2. Technical Report CS-TR-4694, Department of Computer Science, University of Maryland, US, May 2006.

[3] Joint Chiefs of Staff. *The Military Contribution to Peace Support Operations,* 2nd *edition.* Ministry of Defence, UK, June 2004.

[4] Joint Chiefs of Staff. *Noncombatant Evacuation Operations.* United States Forces, January 2007.

[5] Dana Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, Héctor Munoz-Avila, J. William Murdock, Dan Wu, and Fusun Yaman. Applications of SHOP and SHOP2. Technical Report CS-TR-4604, Department of Computer Science, University of Maryland, US, June 2004.

[6] Dana Nau, Héctor Munoz-Avila, Yue Cao, Amnon Lotem, and Steven Mitchell. Total-Order Planning with Partially Ordered Subtasks. 17th *International Joint Conference on Artificial Intelligence*, August 2001.

[7] Stuart Russell and Peter Norvig. *Artificial Intelligence, a Modern Approach,* 2nd *edition.* Artificial Intelligence. Prentice Hall, 2003.

[8] Austin Tate and Ken Currie. O-Plan: the Open Planning Architecture. *Artificial Intelligence*, 52, 1991.

[9] Rik Warren, David E. Diller, et al. Simulating scenarios for research on culture and cognition using a commercial role-play game. In *Proceedings of the 2005 Winter Simulation Conference*, 2005.