MMPM: a Generic Platform for Case-Based Planning Research *

Pedro Pablo Gómez-Martín[1], David Llansó[1],
Marco Antonio Gómez-Martín[1], Santiago Ontañón[2] and Ashwin Ram[3]

[1] Dep. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid, Spain
{pedrop,llanso,marcoa}@fdi.ucm.es
[2] Artificial Intelligence Research Institute
CSIC, Spanish Council for Scientific Research
santi@iiia.csic.es
[3] Cognitive Computing Lab
Georgia Institute of Technology
ashwin@cc.gatech.edu

Abstract. Computer games are excellent domains for the evaluation of AI and CBR techniques. The main drawback is the big effort needed to connect AI systems to existing games. This paper presents MMPM, a *middleware* that supports the connection of AI techniques with games, and specifically geared towards case-based planning. We will describe the features of MMPM, and compare with related systems such as TIELT.

1 Introduction

As several authors have pointed out in the past [1, 5], computer games are excellent domains for the evaluation of AI, machine learning, or Case-Based Reasoning (CBR) techniques; strategy games are well suited for acting as a test bed for all these techniques. The main drawback is the effort needed to connect AI systems to existing games.

A recurrent way used for testing machine learning (ML) techniques is to generate games *traces* containing the complete story of the game sessions played by human experts. Then, the ML algorithms use these traces to infer how to play the game. In a Case-Based Planning (CBP) context, the learning process would extract plans the expert seemed to follow in order to use them in future plays of a game.

These facts were what lead us to the MakeMEPlayME.com idea: an expert plays a game and the game saves a trace that stores every event that has taken place in the session (i.e. actions carried out by game characters) and how the game state has changed afterwards (i.e. changes in entities parameters such as life or position). Traces are the input data for the *learning engine* so it processes these traces and generates a data package which contains the knowledge inferred. This data is what we call a Mind Engine (ME). In this moment the first stage (Make ME) is over. The second stage (Play ME)

* Supported by the Spanish Ministry of Science and Education (TIN2009-13692-C03-03)

consists on confronting the ME to play versus a human player or another ME, imitating the behaviours learned from the expert.

This paper presents MMPM, a *middleware* that supports the MakeMEPlayME.com process. MMPM eases the connection between strategy games and Learning Engines (AI and CBR game playing engines).

In order to prove MMPM, we have connected four different strategy games and a learning engine. The learning engine, known as *Darmok 2* (D2) [9] was based on the MakeMEPlayME.com process, but from an engineering point of view it was tied to the strategy games in such a way that both, games and learning engine, were difficult to reuse. The experience of reengineering them demonstrate to us that MMPM is general enough to support different learning engines and games.

This paper runs as follows: Section 2 is focused on a better explanation of the MakeMEPlayME.com idea and the MMPM middleware architecture. Then, Sections 3 and 4 shows how to define a game domain in a declarative way and how creating Java classes from it to ease the connection between games and Learning Engines. Sections 5 and 6 explained the process of creating a game and a Learning Engine. And the final sections shows some related work and conclusions.

2 General MMPM Architecture

Though theoretically MMPM could be used for different game genres, our main focus is strategy games. Specifically, MMPM expects strategy games with four main components: maps, entities, actions and sensors. Maps are the board where the dynamic elements (or entities) reside. Player and AIs modify the game state using a collection of actions defined in the game rules. When one of the entities performs an action over the map, it may take some time to be completely executed and it can even fail. The other entities are able to inspect the state of the game using some sensors.

From the MMPM point of view, these are the only information it needs to know about the concrete game being played, and it is known as the game domain. Section 3 details how the MMPM users may specify it in a declarative way.

Additionally to games, MMPM assumes the existence of *learning engines* such as D2 [9] (MMPM is well suited for case-based planners, but other learning engines are also possible). As we mentioned above, the essential feature is that they must be able to learn through traces, taken from real player games, in such a way that they should react during a play of a game in the same way as the real player did in the source traces.

From the MMPM point of view, learning engines have two different functionalities: the one who learn from traces (what we call *ME Trainer*) and the one who interprets and executes the previous learned behaviours (*ME Executor*).

With all this in mind, we can state the global experience in MMPM as the one shown in Figure 1. The three steps of a global interaction are:

- An expert plays a game and the game generates a trace that stores every event (*action*), which has taken place during the game, and how the game state has changed due to these actions.

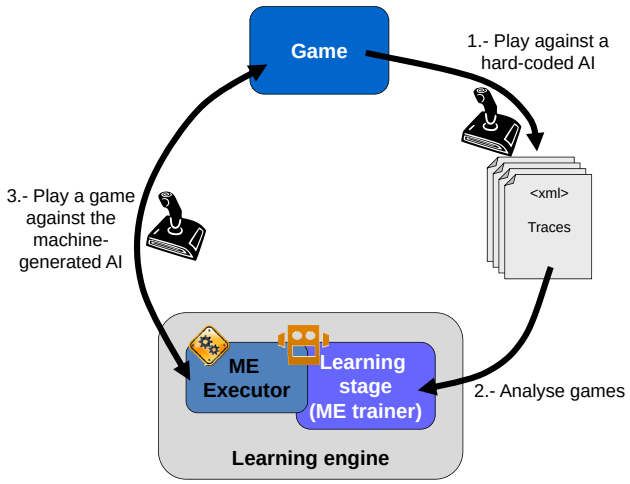

Fig. 1: General Make Me Play Me architecture

- After the expert has played some plays of a game, he decides to generate a new AI based in his strategies. So, he gives one or more traces to the Learning Engine. Its *ME trainer* processes these traces and generates a data package which contains the knowledge inferred, in the form of a *Mind Engine* (ME).
- Once the ME has been created, a user may decide to play against this ME. Thus, the game is launched with a *ME executor* that interprets the ME and it can compete against the user imitating the behaviours learned from the traces. Note that it is also possible to configure the game in such a way that we have two MEs generated from different traces or different learning engines playing against each other.

This process is in fact what we call the MakeMEPlayME.com, and is not specific to MMPM. The task of MMPM is to ease the communication between games and learning engines. Furthermore the goal of MMPM is to allow the addition of new games and new learning engines transparently.

This goal is accomplished by standardizing the three aspects where game and learning engines communicate with each other:

- The trace format in order for every learning engine can understand traces of every game.
- The way in which learning engines can access the game state. In other words, how a game sends the results of sensors used by learning engines to extract information from game state.
- The way in which games receive actions from ME executors to execute them in the world.

Traces are stored using an XML Schema. Figure 2 shows an XML fragment (adapted for simplicity) of the trace of a strategy game called *Towers* (one of the four games

```
<gametrace>
  <entry time="10">
    <gamestate>
      <entity id="3" type="TBase" owner="player1" coor="16,16,0">
        <hitpoints>20</hitpoints>
      </entity>
      <action type="Build" id="4">
        <parameter name="playerID">player2</parameter>
        <parameter name="type">TTower</parameter>
        <parameter name="coor">{304,304,0}</parameter>
        <parameter name="lifeTime">10</parameter>
      </action>
    </gamestate>
  </entry>
</gameTrace>
```

Fig. 2: Trace example of game Towers

mentioned in Section 1). There, the `player2` `builds` a new `TTower`. This example makes clear that the traces depend on the concrete game being traced. In that sense, MMPM XML Schema only specifies the scaffolding of the traces, and it delays the complete trace format until the game domain has been declaratively specified using the ideas described in Section 3.

The communication between games and learning engines is also game dependent. As we will see in Sections 4, 5 and 6, MMPM has a set of tools that automatically generates Java classes from the game domain specification that extends games and learning engines implementations. They should be integrated within the game to generate traces and send the game state to learning engines. As well, they should be used in learning engines to interpret traces during the learning stage (ME trainer), and to receive game states from games, inspect them through sensors and emit game actions in a game session (ME executor).

3 Game Domain

The *game domain* represents all the details about the concrete game and, to some extent, its rules. From the MMPM point of view, it is composed of four kind of elements:

- Entities: they are each piece of interactive content in a game [6]. In other words, they are dynamic objects such as non-players characters or items.
- Actions: all that can be done in the virtual world by players and other entities.
- Sensors: all the aspects that can be measured in the virtual world.
- Goals: the player aspirations in the game.

Each game must provide information about its particularities for all these elements as *declarative knowledge*. MMPM specifies an XML schema that allows game devel-

```
<Entity>
  <Name>TBase</Name>
  <Super>PhysicalEntity</Super>
  <Features>
    <Feature Name="hitpoints" DataType="int" DefaultValue="20"/>
  </Features>
</Entity>
```

Fig. 3: `TBase` entity specification for Towers game

```
<Action name="Build">
  <Parameter name="type" type="ENTITY_TYPE"/>
  <Parameter name="coor" type="COORDINATE"/>
  <Parameter name="lifeTime" type="INTEGER"/>
  <...>
</Action>
```

Fig. 4: `Build` action specification for Towers game

opers to create the domain of the specific target game. The resulting XML will be later used by the learning engines to adapt themselves to each game.

Each entity in the game is composed of a name and the name of the parent entity (MMPM assumes that entities are organized in an entity hierarchy). The description may also include a set of *features* (similar to "fields" or "attributes") that are composed of a name, a datatype and an optional default value. This definition provides an archetype of all the entities of the same type, in a similar way to *classes* in object-oriented languages. When the game state is serialized into traces, the concrete feature values for each entity instance is stored. Figure 3 shows the specification of an entity called `TBase` for Towers.

Actions are the *operators* that players can apply in the virtual world. Figure 4 shows a partial specification of the `Build` action that appeared in the trace on Figure 2, including its name and parameters. When one action is *instantiated* in a trace, real values must be provided for all its parameters.

Under the Make ME - Play ME context, learning engines must analyze when certain actions have been performed in the traces to learn when to use them. In order to help guiding this learning process, and also to help the learning engine understand the relations among actions and how to chain them together, actions specify different kind of *conditions*. For instance, the `Build` action above can only be executed when the coordinates where the building will be built are empty, the building is not blocking the path between player and enemy, and he has enough gold.

Actions in MMPM define the following conditions:

- Valid Conditions: which capture the fact that some combinations of parameters in some actions are not acceptable (e.g. coordinates have to be always inside of the playing area).

```
<Sensor name="Gold" type="INTEGER">
  <Code>IntAttrib(Entity(Type("TPlayer"),player),"gold")</Code>
</Sensor>
```

Fig. 5: `Gold` sensor specification for Towers game

- Preconditions: capture conditions which have to be true for the action to start (e.g. the player has to have enough gold, etc.). The distinction between preconditions and valid conditions is that if a precondition is not satisfied, the player might be able to do something to correct this (i.e. mine more gold) but if a valid condition is not satisfied, there is nothing the player can do to correct it.
- Failure Conditions: MMPM does not assume that actions always succeed, nor that they are instantaneous. Once an action is started, learning engines using MMPM can monitor their execution to determine whether they succeeded or failed. If after the execution of an action the failure conditions are satisfied, it means that the action failed.
- Success Conditions: when these conditions are satisfied, the learning engine knows the action succeeded. Success or failure of an action is thus determined by which conditions get satisfied first: failure or success conditions.
- Postconditions: in addition to success conditions, MMPM allows the definition of additional postconditions, which can be considered as side effects of executing an action.

Thus, action definitions in MMPM differ from traditional action definitions in classical planing languages like STRIPS [3], TIELT [8], or other behavior languages such as ABL [7], in that actions do not just specify preconditions and postconditions. But specify a richer set of conditions, specifically devised to allow the learning engine to reason about actions, and to monitor real-time execution of them.

Action conditions are satisfied or not depending on the world state. MMPM provides a set of *native sensors* that can be combined to specify basic queries about the current world state. These sensors can be parameterized, and they return a concrete value used in the condition expression. Sensors use the same basic datatypes available for entities features and action parameters. MMPM also allows games to specify their own complex sensors using the native ones. For example, in the `Build` precondition mentioned previously, "has gold enough" is described using a new sensor created in the game domain combining the MMPM sensors (Figure 5).

Using the new `Gold` sensor (and some others), the `Build` precondition could be specified in the way shown in Figure 6.

Keep in mind that while entities and actions are serialized into traces, sensors and conditions are static information, useful for the learning engines but that are not stored in traces in any case. Sensors can be also used for communicating the game and the ME executor while playing a game, in a similar way to actions.

Finally, the domain is completed with *goals*. They are *conditions* specified by sensors, boolean and relational operators, that are used by learning engines for guiding their training. As sensors and actions, goals are also static information that is not serialized.

```
<PreCondition>
  <![CDATA[ BuildableCoordinates(coor) && NonBlockingPath(coor)
    && (Gold() >= BuildingCost(type)) ]]>
</PreCondition>
```

Fig. 6: Precondition of the `Build` action specification for Towers game

```
<WinGoal name="WinGoal">
  <Code>
    FLOAT(NumberOfEntities(Type("TBase"),player)) /
      FLOAT(NumberOfEntities(Type("TBase")))
  </Code>
</WinGoal>
```

Fig. 7: `WinGoal` for Towers game

Games must at least specify one `WinGoal` which, when true, indicates the end of the game and the victory of the player (Figure 7). They can also provide non-final goals that act like desired milestones in the game session.

4 Supporting Tools

The communication process between games and ME Executors is not resolved yet. Moreover, to promote developers work adapting games and Learning Engines to MMPM, tools that eases the trace generation in games and trace reading in Learning Engines should be provided.

A generic communication model (e.g. Web Services) could be used to connect games with ME Executors, but this would make the integration quite complex and would slow down the execution of games. To avoid these problems, we assume games and Learning Engines will be implemented using Java is done in other research projects such as Weka [4] and jColibri [2]. Furthermore the Java imposition still allows connections with other languages if the needed bindings were programmed.

Once we assume Java as implementing language, MMPM provides a a tool that automatically generates classes from the domain specification mentioned in Section 3. For instance, Figure 8 shows part of the `TBase` entity class automatically generated from the XML domain (Section 3). Apart from translating information described in the domain, the tool adds methods for serializing and deserializing traces in the XML format mentioned in Section 2.

All these generated classes are based on several general classes that conform the MMPM middleware. There are superclasses for entities, actions and sensors and support for MMPM native sensors and types. Binding our middleware to Java has an additional advantage: when there is not a way to define a new game sensor using the MMPM native sensors, it can be defined using plain Java in the XML domain. Although `Gold` sensor (Figure 5) was able to be specified using a native expresion,

```
public class TBase extends PhysicalEntity {
  public TBase(String entityID,String owner) {
    super(entityID, owner);
    _hitpoints = 20;
  } // Constructor
  ...
  protected int _hitpoints;
  ...
} // class TBase
```

Fig. 8: Fragment of the java code generated for the `TBase` entity

`NonBlockingPath` sensor that `Build` precondition used (Figure 6) cannot be specified in this way. Nevertheless it can be defined in the XML domain in Java code as in Figure 9.

```
<Sensor name="NonBlockingPath" type="BOOLEAN">
  <Parameter name="coor" type="COORDINATE"/>
  <Code language="java">
    <![CDATA[
      boolean[] blocked = new ...
    ]]>
  </Code>
</Sensor>
```

Fig. 9: Java sensor example

As we will see in Section 6, this alternative and non declarative way for defining sensors is not a problem for Learning Engines. They are written in Java too, so they can evaluate these sensors executing them.

5 Developing a Game for MMPM

First of all, game developers must specify the game domain as explained in Section 3 and they should create and generate the Java classes (Section 4). These classes will be very useful to generate game traces (the first stage in the MakeMePlayMe.com process shown in Figure 1) and to let learning engines execute actions within the game (the third stage).

If game developers wanted to adapt a previously created game, they would need to write methods to convert the original game state representation into objects of the MMPM classes and vice versa.

On the other hand, in case the game was not developed yet, the auto generated classes could be even used as the skeleton of the logic of the game. Every entity, action

and sensor class would be generated with its features and ready to be serialized, so game developers should save a lot of time using them as part of the game itself. They would have to implement how actions modify the game but a lot of work would be already done.

During a play of the game, traces must be generated. So, the game must store its game state and produced actions every certain period of time and then, the trace has to be sent where the one who launched the game specified. MMPM provide a *tracer*, a set of Java classes that the game invoke periodically with the MMPM game state objects to be stored.

The other connection that must be made is between the game and a ME executor. First of all, MMPM has a ME executor factory that games may use to create the machine-generated AIs. Afterwards, the game invoke them using MMPM game state objects to produce the MMPM actions according to the strategy inferred by the Learning Engine. Game code will translate these game-domain actions into the implementation-dependent actions that are used to perform them into the game.

6 Developing a Learning Engine for MMPM

A Learning Engine is compound of two different systems: a ME Trainer and a ME Executor. A ME Trainer is an off-line system that does not interact with games but with their game traces. From the MMPM point of view, and to ensure independency between systems, the Learning Engine is a black box that receives game traces and creates a ME from them. In order to manipulate game traces easily, Learning Engines should use the auto-generated domain classes. Furthermore, if the Learning Engine uses MMPM super classes as native classes, it can make use of their serialization to store MEs in the ME Trainer system and to load them again in the ME Executor.

On the other hand, a ME Executor is bound to games and it must react quickly. It receives game states and reacts returning actions to be executed in the game. The machine-generated classes must be used because game states are given, and actions must be returned to games, as objects of these classes. Moreover, using these classes is the only way that Learning Engines have to evaluate sensors and action conditions, which are, somehow, the rules of the game. Action conditions should be checked before returning them to the game in order to know if their execution in the actual game state makes sense.

7 Related Work

The need for a system such as MMPM that ease the complexity of developing Learning Engines and connecting them to games was identified in the past by Molineaux and Aha [8], who designed the TIELT system to address this problem.

Our goal is to encourage progress in this area and offer an alternative to TIELT, which is more suitable for certain kind of games. The main differences between TIELT and MMPM are the problems they try to solve and the formalism to define some of the game domain knowledge. For instance, while TIELT also attempts at solving the problem of performing experiments, MMPM only attempts at bridging AI components

with games. Another big difference between TIELT and MMPM is the way actions are specified. For TIELT, an action is defined by a set of capable roles (who can execute this action), a set of preconditions, and a set of state changes (that define how the state will change after executing this action). The state changes are defined as a script, which defines the state transformation done by the action. MMPM has a richer action definition language, allowing: valid conditions (restrictions on the input parameters), preconditions, success conditions (conditions that if satisfied we can consider the action succeeded), failure conditions (conditions that if satisfied after the action was initiated we know that the action will never succeed), and additional postconditions (with a similar meaning to the state changes in TIELT). This action definition was defined with the goal in mind of allowing the AI systems to easily monitor the execution of actions at run-time, which is hard to achieve using TIELT.

8 Conclusions

This paper has presented MMPM, a *middleware* that supports the connection between games and learning engines, promoting the development of machine learning techniques in games. In that sense, MMPM recalls some other research projects such as TIELT.

We have also adapted an existing Learning Engine known as *Darmok 2* (D2) to be used with MMPM. D2, that uses Case-Based Planning to create machine-generated AIs from expert traces, has been successfully used with up to four different games that were also adapted to be used with MMPM.

Currently we are working on extending the game domain to allow more complex actions (or operators).

References

1. M. Buro. Real-time strategy games: A new AI research challenge. In *IJCAI'2003*, pages 1534–1535. Morgan Kaufmann, 2003.
2. B. Díaz-Agudo, P. A. González-Calero, J. A. Recio-García, and A. A. Sánchez-Ruiz. Building CBR systems with jCOLIBRI. *Special Issue on Experimental Software and Toolkits of the Journal Science of Computer Programming*, 69(1-3):68–75, 2007.
3. R. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intellicence*, 2(3/4):189–208, 1971.
4. M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, June 2009.
5. J. E. Laird and M. van Lent. Human-level AI's killer application: Interactive computer games. In *AAAI 2000*, pages 1171–1178, 2000.
6. N. Llopis. *Introduction to Game Development*, chapter Game Architecture, pages 267–296. Charles River Media, 2005.
7. M. Mateas and A. Stern. A behavior language for story-based believable agents. *IEEE intelligent systems and their applications*, 17(4):39–47, 2002.
8. M. Molineaux and D. W. Aha. Tielt: A testbed for gaming environments. In *AAAI*, pages 1690–1691, 2005.

9. S. Ontañón, K. Bonnette, P. Mahindrakar, M. Gómez-Martín, K. Long, J. Radhakrishnan, R. Shah, and A. Ram. Learning from human demonstrations for real-time case-based planning. *IJCAI-09 Workshop on Learning Structural Knowledge From Observations (STRUCK-09)*, 2009.