IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 23, 2019, accepted December 17, 2019, date of publication December 30, 2019,
date of current version January 8, 2020.

Digital Object Identifier 10.1109/ACCESS.2019.2962969

XML-Based Video Game Description Language

JORGE R. QUIÑONES[1] AND ANTONIO J. FERNÁNDEZ-LEIVA[2]
[1]Department Lenguajes y Ciencias de la Computación, Universidad de Málaga, 29071 Málaga, Spain
[2]ITIS Software, Department Lenguajes y Ciencias de la Computación, Universidad de Málaga, 29071 Málaga, Spain

Corresponding author: Antonio J. Fernández-Leiva (afdez@lcc.uma.es)

This work was supported in part by the Spanish Ministry of Economy and Competitiveness (MINECO) DeepBio under Grant TIN2017-85727-C4-1-P, and in part by the Universidad de Málaga.

ABSTRACT This paper presents the XML-based Video Game Description Language (XVGDL), a new language for specifying Video games which is based on the Extensible Markup Language (XML). The proposal is portable and extensible, and allows games to not only be defined at engine level but also includes specific features that can lead the game design process whilst simultaneously reducing the gap between game specification and its corresponding game implementation. XVGDL is as generic as possible, making it possible to describe different genres of games. This paper focuses on presenting the basis of the language. The paper describes the syntax as well as the components of XVGDL, and provides examples of their use. Defining games via XML structures provides all the advantages of the management of XML files and opens up interesting lines of research. Our proposal provides a number of novel features. So, XVGDL game definitions can be managed as any other XML file, which means that it can be automatically handled by any XML file management software. Another interesting feature is that XVGDL can specify game components (e.g., game Artificial), in-game processes (e.g., the procedural generation of maps) or in-game events (e.g., the checking of the conditions to end a game match) via the association with external (possibly non-XML) files. Moreover, XVGDL files can be easily validated as any XML file what means that validations against a particular Document Type Definition (DTD) or XML Schema Definition (XSD) are possible. In addition, the paper presents a first prototype implementation of a (text-based) interpreter that allows XVGDL game specifications as a playable game to be executed. This tool not only validates our proposal but also represents a first step towards smoothing the path to obtaining an executable version of a game from its game specification.

INDEX TERMS Video game description language, extensible markup language, XML, game design, game tools.

I. INTRODUCTION

The importance of a Video Game Description Language (VGDL) has been reported in the literature. Its use is especially interesting in General Video Game Playing (GVGP) where the objective is to create autonomous, automated agents capable of learning to play previously unknown games without human intervention and just by being told the rules of a game [1]. GVGP has a strong influence on many Artificial Intelligence (AI) areas, especially in Non-player character (NPC) behaviour learning, search, planning, and the employment of games as AI benchmarks [2].

Within the community of Artificial and Computational Intelligence in Games, one of the most important and valuable objectives is the definition of a language that allows video games to be specified. Formally, this language should provide structures to define the main components of a video game such as rules, mechanics, events, physics, user interaction, multimedia elements or narrative.

Having a VGDL to specify games can aid the human understanding of the features and mechanics of the game, and can also simplify the task of implementing general video game agents. In addition, specifying different games with the same VGDL can promote the attainment of a general evaluator to measure the quality of an automated agent to play the games described via the VGDL, in a general way.

Also note that, the specification of video games via a VGDL may be compiled with a compiler, specifically built-in for the VGDL. This compiler could provide numerous opportunities and paves the way for automatic game generation [3]. In fact, this is a desired goal, that is to say, giving a game

The associate editor coordinating the review of this manuscript and approving it for publication was Mario Luca Bernardi.

specification as input to a compiler with the aim of producing an executable version of the game. As indicated in [3], ''Implementing such a compiler could provide numerous opportunities; users could modify existing games very quickly, or have a library of existing implementations defined within the language (e.g. an Asteroids ship or a Mario avatar) that have pre-existing, parameterised behaviours that can be customised for the users' specific purposes''. Moreover, finding a standard VGDL would provide a significant contribution because researchers can concentrate their efforts on the implementation details of the aforementioned desired compiler.

Thus far, a number of different VGDLs have been proposed (see Section II). However, whereas all of them provide interesting features, most of them have been defined to cover specific games or genres of games. In fact, the majority of these VGDLs were defined from scratch and their syntaxes specifically adjusted to their specific objectives. This represents a restriction, not only on developing a standard compiler but also on convincing the community about the validity of compilers to define games generically.

This paper presents a new VGDL that provides features not present in other VGDLs. This is the main contribution of this paper. In common with other VGDLs, our proposal includes components to define game elements, mechanics, rules, and scheduled events. In addition, XVGDL also allows specific design components to be described such as game maps, screen configurations and viewports, game renderers or game states. It can also be easily extended to support some other features like, for example, multimedia elements.

The main difference with other existing VGDLs is that our proposal is based on the well-known Extensible Markup Language (XML). This distinction implies important advantages over existing VGDLs. First, the fact that our proposal is substantiated in a well-known markup language makes it attractive for the community as it increases its potential to be generalised and accepted as a standard VGDL. Second, XVGDL may be viewed as a general-purpose language in the sense that the games described via XVGDL are XML files and, thus, are portable and can evolve naturally with the XML language. Third, the definition of a game via XVGDL is an XML file, and as a consequence, can be directly handled by any existing XML management software. Fourth, we have defined a (publicly available) XML Schema Definition (XSD) for our XVGDL, termed XVSD (see [4]). The consequence is that any XVGDL game specification can be easily validated according to these schemes.

A second contribution of this paper is to demonstrate that our proposal also aims to reduce the gap between the specification of a game and its implementation (i.e., an executable version of the game according to its specification). The paper presents a prototype implementation of a (text-based) interpreter that receives as input an XVGDL game description and allows executing a playable version of the game.

The paper is structured as follows: Section II discusses related work and compares our proposal with other VGDL languages. Section III describes all the main components and features of the XML-based video game description language (XVGDL). Then, Section IV presents the XVGDL Game Engine, a first prototype of an interpreter of XVGDL game specifications. The paper ends with our conclusions and mentions to some lines of future research.

II. RELATED WORK

In addition to aforementioned approaches, this section discusses other Video Game Description Languages.

Existing research papers have expounded initial ideas and established guidelines of what a VGDL should define and how to incorporate video game elements into written configurations (i.e., the game specification written in a specific VGDL) [3]. The importance of finding a formal way that allows specifying video games concisely is, nowadays, a well-accepted approach within the AI community [1], [3].

One can find, in the scientific literature, approaches which primarily focus on theoretical models, describing games using mathematical models, like *General Video Game Playing* [1] or *Epistemic GDL* [5]. It is also worth mentioning the work presented in [3] (referenced in Table 3 as TWVGDL) which establishes the basis for a generic VGDL [6].

If we consider just games (and not video games), GDL (i.e., the Game description Language) can be considered the standard of General Game Playing (GGP) for describing games (even dealing with both randomness and imperfect information) [7]. The GDL basically represents a mathematical notation similar to the syntax and semantics of logic programming that cannot be directly extended to cope with video games (because, for instance, it does not provide structures or components to describe multimedia elements or user interaction).

It is also difficult to find proposals of VGDLs whose game specifications can be compiled (or interpreted) into an executable, and that includes the use of standard tools/languages. Some experiments have been reported in the literature for obtaining a playable video game from the game specification written in a concrete VGDL. This is the case of PyVGDL, a VGDL based on the programming language Python [8]. The approach represents a good starting point to compile game specifications, but it is developed in a very specific way for describing video games so its use is very limited. Other interesting proposals suggest the integration with commercial game engines. This is the case of Casanova, a declarative programming language that allows the integration with the well-known Game Engine Unity [9] and which can also be used with other engines and libraries [10], [11]. However, Casanova is more oriented to developing games and not video games. In general, we can say that the existing implementations of compilers for VGDLs are usually *closed* and developed in a very specific way for describing video games. In other words, they are tied to certain features or specific games and therefore are not transferable to different tools or engines (like Unity for example).

Apart from the aforementioned PyVGDL, one can find other similar approaches to XVGDL that are associated with high-level programming languages such as Ludi [12], but this is restricted to 1-player board games.

It is worth mentioning that there are also existing approaches, such as Ludocore [13] that help designers to develop rules and their relations, contributing with a flexible game engine for logical games, able to be applied to both GDLs and VGDLs. It is in the scope of XVGDL to establish the basis to allow a game engine to interpret and run games making Ludocore an excellent source of information related to future XVGDL research.

In the following section, we describe XVGDL and its main components.

III. THE XML-BASED VIDEO GAME DESCRIPTION LANGUAGE

Based on VGDL and work already cited in this paper ([3], [8], [14] among others), XVGDL is a new language for describing games that is portable, extensible and covers as many aspects of a game as possible, including those concerning game design. The initial idea is to use XVGDL for research purposes, but, thanks to its features and capabilities it can also be used to create a basis for games to be exported to commercial game engines.

A. DESCRIBING A GAME USING XVGDL

In addition to the aforementioned features that XVGDL provides, this language also has the following:

- It includes all the principal features of current VGDLs already described in the video game research community and not only includes the game description itself but also has special tags and values to be taken into account by the game engine.
- It is designed to be as generic as possible in order to allow a wide range of games to be defined. As will be shown in this paper, the language offers the possibility of specifying the association of in-game tasks with external files that will handle the execution of these tasks. This is a powerful mechanism that links the game specification with a possible implementation. So, it allows, for instance, the (perhaps procedural) generation of game maps/levels, the control of (perhaps automated) game AI, the activation of events, the checking of parameters during a match of a game, just to give a few examples). In fact, this feature gives designers the option of creating different variations of a game by programming external processes that affect the game components (e.g., game layout or game AI).
- All main features related to game design – such as game maps, game renderer, game rules or game mechanics – are included in order to describe games. Covering all those aspects ensures that a well-designed XVGDL game may be compiled (or interpreted) into a playable version by a game engine able to run XVGDL games or, in a further step, export XVGDL games to any of the commercial engines' formats.

B. EXTENSIBLE MARKUP LANGUAGE ADVANTAGES

The Extensible Markup Language, XML, is a general-purpose markup language developed by the World Wide Web Consortium (W3C) with the goal of storing data in a readable form [15]. XML is proposed as a standard for the exchange of structured information between different platforms, including the Internet, but not necessarily for it. It can be used in databases, text editors, spreadsheets, etc. Being of general-purpose, and unlike other markup languages, XML is not totally predefined, in the sense that users can define new tags for their own applications.

Our proposal of developing a VGDL based on XML is based on the power and potential of XML and its simplicity, generality, and usability across different platforms and environments. The XVGDL language is based on the standard XML language, so it inherits its main features to make it extensible, maintainable, and portable. It is also flexible and open to any required future upgrades to improve the current definition.

Games described in XVGDL are XML files and, therefore, they can make use of libraries dedicated to managing XML files. In addition, the XVGDL game specifications are not restricted to any specific programming language or operating system.

In addition, XML can make use of data structures, which is good for our purposes, as XVGDL allows not only game rules and mechanics to be defined but also game objects and elements. These data structures and their relationships can be considered as data in themselves. Moreover, XML allows interrelating of these data structures. This feature is extremely useful in issues like defining collision rule actions, where one considers two kinds of objects. Here, it is possible to make a reference to the objects under collision using the object-defined name, as shown in Figure 1.

C. XVGDL VALIDATION

One of the most important steps when a designer writes a game described with a VGDL is to validate whether or not it is well-written in the language being used, which means checking whether or not it follows the language syntax. XVGDL files can be easily validated as any XML file which means that it allows validations against a particular Document Type Definition (DTD) or XML Schema Definition (XSD). This provides the opportunity to define the structure and valid building blocks of an XML document, taking the following key concepts into account:

- The elements and attributes that can appear in a document.
- The number of (and order of) child elements.
- Custom and standard data types for elements and attributes.

```
1  <!-- Players Definition -->
2  <players ...>
3  <player name="pacman" .../> // some more properties are included
4  </players>
5  ...
6  <!-- Defining the Objects present in the game (NPCs, Obsctacles, projectiles, structures, collectibles, etc... -->
7  <objects>
8  <object name="wall" type="wall" .../> // some more properties are included
9  </objects>
10 ...
11 <rules>
12 <rule name="ghostCatchPacmanLivesDown" type="collision">
13 <ruleAction objectName="pacman" result="livesDown" />
14 <ruleAction objectName="ghost" result="" value=""/>
15 </rule>
16 </rules>
```

FIGURE 1. Example XVGDL referencing defined objects.

- Default and fixed values for elements and attributes, including enumeration of elements for specific data types.

We have worked to provide an XML Schema Definition for XVGDL (XVSD) which is available in [4]. This XVSD can be employed to make sure all XVGDL files are valid (as an XML file) and correct (according to this schema definition). In order to upgrade the XVGL language, it is mandatory to update the schema with the required modifications.

It should be evident that our XVSD is a previous step for obtaining a compiler for the definitions of video games written in XVGDL.

D. GAME SPECIFICATION AND ELEMENTS IN XVGDL

Game specifications are XML files in which game components are defined as XML components. Figure 2 illustrates graphically the XVGDL components and their relations. An XVGDL specification contains the description of a game, and it usually starts with a standard XML declaration in the form:

```
<?xml version="version_number"
    encoding="encoding_declaration"
    standalone="standalone_status"
 ?>
```

In general, the rest of an XVGDL game description (also termed as specification throughout the paper) follows the following code scheme:

```
<gameDefinition>
  <!-- Renderer configuration -->
    ...
  <!-- Timeout configuration -->
    ...
  <layout> ... </layout>
  <map ...> ... </map>
  <controls ...> ... </controls>
  <players ...> ... </players>
  <objects> ... </objects>
  <events> ... </events>
  <rules> ... </rules>
  <endConditions> ... </endConditions>
  <gameStates> ...</gameStates>
</gameDefinition>
```

Thus, the specific components managed in the game description can be identified as the content enclosed between the start-tag `<gameDefinition>` and the end-tag `</gameDefinition>`. In a general way, the list of specific game components managed in XVGDL can be classified as: properties, layout, maps, controls, players, objects, events, rules, and end conditions. Note that XVGDL is a language in itself and different implementations of interpreters, engines and tools are responsible for interpreting all of the tags.

We now provide details of each of the aforementioned XVGDL components. To clarify them, we define the classic Pacman video game [16] in XVGDL to illustrate the main components of the language. Pacman has been chosen for the case study because it is a well-known game, already used in several studies for generating content, applying any kind of AI to enemies or players [17]–[20] and it is also simple enough to understand how to configure and prepare XVGDL for a complete simulation. A full XVGDL configuration for the Pacman game can be found in [21], and part of it is included in Appendix. Moreover, we provide the full XVGDL description of standard versions of other well-known games (i.e., Breakout, Space Invaders and an example of a first person shooter) in [22]. In the following explanations we refer to certain lines of the Pacman XVGDL specification, shown in its entirety in Appendix, for a clearer comprehension of the XVGDL components.

1) PROPERTIES

A list of properties can be set for a game. Each property is set by the tag `property` followed by a number of attributes in the form `<property attribute1 attribute2 />`. For instance, see in Appendix, the definition of a property with the name and value of attribute *timeout* in Line 8 and its use in Lines 71–72 to declare one of the end conditions of the Pacman game.

The definition of properties is completely open to designers' specific needs, so properties with any name and value can be defined. An XVGDL Game Engine or interpreter implementation could access any of the defined properties in the context and update them in runtime if needed.

2) LAYOUT

This component allows the layout of the game to be declared. It is defined by the content enclosed between start-tag

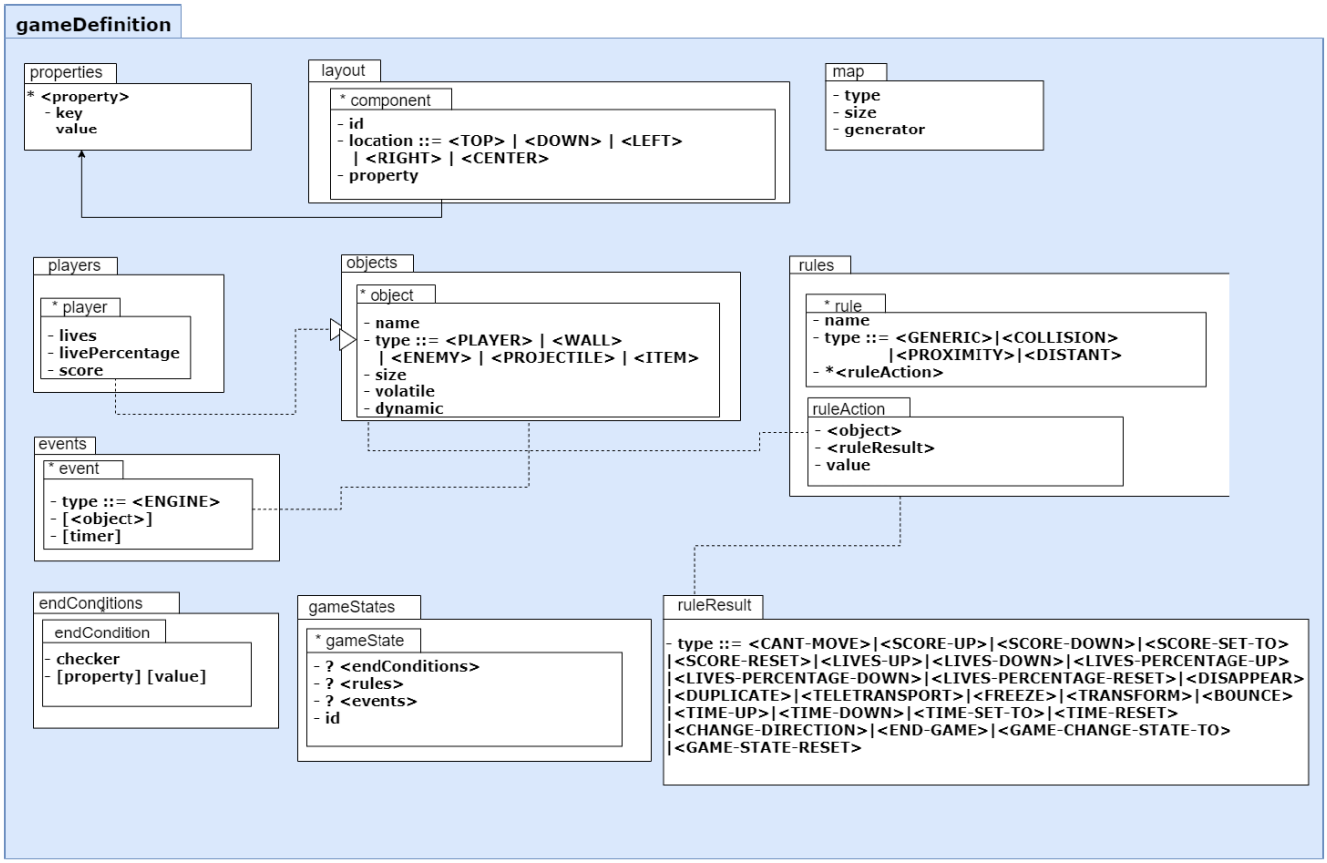


FIGURE 2. XVGDL components.

`<layout...>` and the end-tag `</layout>`, and it used to specify the visual layout of the game.

XML elements therefore can be used to describe specific components containing game state properties or elements that should be presented in the game view. The syntax employed for the layout components is:

```
<component attribute1 ... attributeN>
   ....content
</component>
```

The current version of XVGDL allows elements to be placed in four positions, namely, `top`, `left`, `right`, and `bottom`. The position labelled as `center` is reserved for the game view itself. In each one of these positions, different layout components can be set. See for instance Lines 10–20, in Appendix, where the game score and the highest score (obtained in previous games) are configured in the `top` of the screen, whereas player lives and player items are set in the `bottom` position. Note that other components such as life percentage may be integrated in a similar way.

At the same time, each component may have its own rendering specification using the attribute `renderer`. See for instance Line 6, in Appendix where the renderer configuration is defined in another XML file. This is precisely one of the added values of XVGDL, the possibility to provide the definition of a number of game components via external files (not only in the XML format as demonstrated later in this paper.

3) MAP

This component is employed to define the main properties of the game scenario. The syntax employed to identify this component in the XVGDL file is the following:

```
<map attribute1 ... attributeN> </map>
```

This component admits a number of attributes in the form `tag = value`. So, the following tags and values may be used in the current version of XVGDL:

- The tag `type` can take the values `2D` or `3D` to specify the dimensional nature of the game;
- The tags `sizeX`, `sizeY` and `sizeZ` can take a numerical value specifying, respectively, the length of the map in each of the three coordinate axes.
- Other tags like `toroidal` can be set with a Boolean value (i.e., `true` or `false`) to indicate additional features of the game map (in this case, a true value indicates the toroidal nature of the map)
- The tag `file` can be assigned to an external file (i.e., a file that is not part of the game specification written in XVGDL) that can provide information to establish the initial distribution of game objects in the map.

Moreover, it enables other engines/tools to be employed to read that file and represent the map according to that configuration.
In addition, the tag `generator` can also be used to be assigned to an external file where the distribution of game objects may be done automatically (perhaps procedurally or by any other method). This option is particularly useful for experimentation as it allows maps to be created by external code.

See for example Lines 29–31, in Appendix where for the game Pacman, a 2d non-toroidal map of size 20 × 40 is defined. The initial placement of objects is left to the execution of an external random generator. In this example, the generator places all game objects randomly.

4) CONTROLS

This component has been added to be used as a link between the game specification and its possible implementation. It allows the game designer to define the mechanism for user interaction in the specification itself (the current version of XVGDL supports the employment of values associated with a keyboard). The syntax employed to identify this component in the XVGDL file is:

```
<controls>
  <control attribute1 ... attributeN \>
  ...
  <control attribute1 ... attributeN \>
</controls>
```

where each component `<control . . . />` is associated with a mechanism of data input (e.g., keyboard or gamepad, for instance). The attributes take the classical form of `tag = value`. The current version of XVGDL admits the following tags for the keyboard: `left`, `right`, `up`, `down`, `forward`, `backward`, `rotateXleft`, `rotateXright`, `rotateYleft`, `rotateYright`, `rotateZleft`, `rotateZright`, `jump`, `fire`, and `special`. The value associated with each tag (if presented as an attribute in the game specification) can be any ASCII code that identifies the key associated with the action described in the tag. The degree of rotation or advance in the movement keys can be configured in the rendered configuration as explained previously.

See for example Lines 25–27, in Appendix where for the game Pacman, the movement actions have been associated with the arrow keys.

5) PLAYERS

This component provides information about the players who will take part in the game. The syntax employed to identify this component in the XVGDL file is the following:

```
<players number="1" maxnumber=...>
  <player name = STR score =...>
  ...
  <player name = STR score =...>
</players>
```

where STR denotes a text string. Basically, it declares the number of players, and other information specific to each player such as player name, the initial number of lives, initial score or life percentage. See for example Lines 33–35, in Appendix where we have declared one human player with an initial number of 3 lives.

A property denoted as `ai` can also be used to assign a specific Artificial Intelligence (probably implemented externally) to the player(s) in a similar way as done with the aforementioned map generator. This provides a useful way to execute the game autonomously without any human intervention, which is perfect for issues related to the game design or debugging.

Players declared in this part have the status of objects (as explained below) in the game.

6) OBJECTS

This component is used to declare the nature and properties of each object that will appear in the game, apart from the players already declared, as described. The syntax employed to identify this component in the XVGDL file is:

```
<objects>
  <object name=STR type=... dynamic=.../>
  ...
  <object name=STR type=... dynamic=.../>
</objects>
```

where for each XML object' attributes can be used to declare its name, its type, whether it is dynamic or static, its volatility, number of instances, etc.

With respect to the `type` of the objects, the current version of XVGDL supports the following values:

- `player`: The object will be associated with one of the players declared in the player component.
- `enemy`: The object represents a player enemy which usually causes harm to the player in some way.
- `projectile`: The object represents any kind of player or enemy-thrown weapon. This element may not exist at the start of the game but can appear during the game depending on the activation of events. Think, for instance, when one presses the `fire` key (see below).
- `wall`: The object is a non-traversable map. For instance, the object can be used to establish map limits.
- `item`: The object represents an item that can trigger a concrete event, applying rules and mechanics in the same way as any other kind of object.

Other properties such as the initial position of an object, the `speedFactor`, `size`, whether it is `dynamic` or `volatile`, and the number of instances of any object can also be declared in the object declaration. For every instance of an object, engines/tools should create an internal identifier that differentiates one from another.

An attribute termed as `ai` can also be used to assign an artificial intelligence to govern the behaviour of an object. This is particularly useful for enemies, projectiles, and also players (as already mentioned). As with other components in XVGDL, the behaviour of the artificial intelligence can be left to an external file. However, the current version of XVGDL provides the following built-in AI definitions:

- `chase_player`: the object chases the player around the game map.
- `go_to`: the object tries to move to a concrete position on the map.
- `go_to_object`: the object tries to reach the position of another concrete object. It is thus a more generic version of `chase_player`.
- `random_movement`: the object moves randomly around the map.
- `path`: the object moves through a path of positions on the map.

An attribute termed as `size` can also be used to assign object sizes. Note however that these values cannot be considered as pixels as this depends on a number of factors, such as the designer's criteria. For instance, in a game based on square cells, size units can represent one cell, while for other kinds of games, size can represent a factor to calculate the pixels of a concrete object. In this sense, and as shown previously, XVGDL provides the possibility of configuring sizes and XVGDL interpreters will be in charge of managing those sizes accordingly.

As an example, see Lines 37–44 in a classical XVGDL Pacman specification shown in Appendix. See Lines 39–40 where dynamic and volatile objects called *ghost* of type `enemy` are associated with distinct built-in AIs. So, Line 39 declares two instances of a ghost that will chase the player, and Line 40 declares another two instances of a ghost that will move randomly. Other objects, including non-volatile and static (and that cannot be traversed) like the walls (Line 38), or dynamic and volatile like smalldots (Line 41), bigdots (Lines 42), or cherries (Line 43), are defined in the game specification.

Also note that in this part, the objects are declared but not the game mechanics, which are defined in the rule section. So, big and small dots are objects that Pacman must collect in order to win the game, whereas the cherry is an object that is displayed on the map and affects Pacman in several different ways such as awarding extra ability, points or lives.

7) EVENTS

This component is employed to declare game events, which occur mainly during game execution in response to some interaction with the player (though not necessarily so). Examples of game events are associated with the action(s) executed when a user presses keys. In this case, for instance, the player moves, shoots or takes any other action. The action might also affect the game in other ways, as for instance pausing its execution. Some other events can be configured to occur during the gameplay programmatically (i.e., by coding). For instance, items appearing on the screen in a random position or enemies being created at a concrete point of the map from time to time. Those kinds of events can be easily configured simply by specifying the `class` implementing the event action. As an example, consider again, a classical Pacman specification in XVGDL in Appendix;

TABLE 1. Types of game rules supported in current version of XVGDL.

Rule Type	Description
`generic`	Rule that may not require any objects to be present. Supported to allow events to be configured as rules.
`collision`	Rule that is triggered when objects collide in the game. For each one of the objects, a rule action should be defined in order to apply a consequence to each of the objects involved in the collision.
`proximity`	Rule that is triggered when objects are next to each other. Attribute `value` defines the concrete distance to be considered. Objects separated less than or equal a given distance triggers rule actions
`distant`	The opposite of `proximity`. Objects separated equal or more than a given distance (set in the attribute `value`) triggers the rule actions

Lines 47 and 48 declare two events to spawn cherries and ghosts, and each number of configured time units (to be interpreted by the XVGDL interpreter), respectively.

Distinguishing between `consumable` or `not-consumable` events is a key aspect for designers. While a key press is consumable (once executed, the event is not considered again), a configured `consumable` event lasts forever while the game loop is being executed. It is the responsibility of the game engine (or XVGDL interpreter) to manage those events according to their type. Those consumable events should be processed just once by the interpreter while those that are not consumable must be kept, to execute them in each game loop execution.

8) RULES

This component allows the mechanics of the game to be defined via its game rules. Game rules define the general behaviour of a game when an event occurs involving two or more objects already present in the game state. For instance, a rule is configured to explain what happens when a player gets an item or a player is hit by an enemy. The syntax employed to declare rules in the XVGDL file is:

```
<rules>
  <rule name=STR type=...>
   <ruleAction objectName=STR result=...value=.../>
   <ruleAction objectName=STR result=...value=.../>
  </rule>
  ...
</rules>
```

Each rule is defined by declaring its name (i.e., a text string STR, as "eatSmallDot", for instance), its type (i.e., `generic`, `collision`, `proximity`, or `distant`, the types available in the current XVGDL version as shown in Table 1), and one or more actions that will be executed upon the activation of the rule. The activation of a rule launches the execution of its actions. The number of actions associated with a game rule depends on both the type of the rule and the number of objects involved in its activation. Each action is declared in the form `<ruleAction.../>`) and its execution can influence the game in many diverse ways (and this is declared in the properties `result` and `value`). For

TABLE 2. Results that can be used in the rule actions in current version of XVGDL.

	Result Type	Description	Value
1	`cant-move`	Object can't move to the intended position	
2	`score-up`	Player's score up	Attribute `value` represents the amount of score to be incremented
3	`score-down`	Player's score down	Attribute `value` represents the amount of score to discounted
4	`score-set-to`	Player's score set to a specific amount	Attribute `value` represents the new player's score
5	`score-reset`	Player's score reset to 0	
6	`lives-up`	Player's lives up	Optional attribute `value` represents the number of lives to be incremented
7	`lives-down`	Player's lives down	Optional attribute `value` represents the number of lives to be discounted
8	`lives-percentage-up`	Player's live percentage incremented	Attribute `value` represents the amount of percentage to be incremented
9	`lives-percentage-down`	Player's live percentage reduced	Attribute `value` represents the amount of percentage to be discounted
10	`lives-percentage-reset`	Player's live percentage reset to 100	
11	`disappear`	Object disappears from game	
12	`duplicate`	Object is duplicated in game	
13	`teletransport`	Object is moved in map to different position	If no value is configured, position is generated randomly. Attribute `value` specifies an `x,y,z` position in map. The reference to any other object in map can be set as well.
14	`freeze`	Object is frozen, by default 1 second	Attribute `value` specifies concrete number of seconds.
15	`transform`	Object is transformed into a new object specified in value attribute	Attribute `value` represents the new object class.
16	`bounce`	Object bounces	Attribute `value` specifies the size of bouncing in format `x,y,z`.
17	`time-up`	Game timeout is incremented	Attribute `value` specifies number of seconds.
18	`time-down`	Game timeout is decreased	Attribute `value` specifies number of seconds.
19	`time-set-to`	Game timeout is set to a concrete value	Attribute `value` specifies the new timeout.
20	`time-reset`	Game timeout is reset to its initial value	
21	`end-game`	Game end condition specified by a game rule action type	
22	`change-direction`	Object changes its direction	It is applied randomly (if no `value` is defined) or with a concrete direction vector inf format `"x,y,z"`. The keyword `invert` can be used to invert the current direction vector
23	`game-state-transition`	Set the game to a specific game state	All elements defined under the specific game state are applied
24	`game-state-reset`	Set the game to the default specific game state	All elements defined for the default game state are applied
25	`set-fire`	Set the player fire to a concrete object that should be a valid defined object	
26	`set-special-fire`	Set the player special fire to a concrete object that should be a valid defined object	

instance, a rule action can affect the state of the objects (e.g., they might disappear), or it might increase the value of the game score, just to give a couple of examples.

A rule action defines how the rule exactly affects a given object when the rule that contains the rule action is activated. This means that there is a rule action defined specifically for each object affected by the activation of the rule. In the rule actions, the property `result` identifies the action that is executed when the rule is activated, and the property `value` (if declared in the rule action) defines the intensity of the action (i.e., how the action affects the object).

Table 2 shows all the results (i.e., actions) that can be used as a rule action in the current version of XVGDL. The first column identifies the action to be assigned as a result, the second column describes the consequences of its execution, and the third column explains how the value associated with the action in the rule action affects the game (or the object). For example, when a player gets an item, from the point of view of the player, its score might be incremented by an amount n (and this is associated with a result of type `score-up` with value n). However, from the point of view of the item, this might disappear, which is associated with a result type `disappear`).

Table 2 describes all supported game rule types and actions in the current versions of XVGDL. Note that, although the XVGDL schema currently defines certain values, it can be extended for future needs.

As examples, see again the Pacman specification in XVGDL in Appendix. Lines 51–68 define the mechanics of the game via the declaration of four basic game

rules. So, rule *eatSmallDot* (resp. *eatBigDot*) defined in Lines 52–55 (resp. Lines 56–59) indicates that, when Pacman (i.e., the player object) collides with (i.e., eats) a `smallDot` (resp. `bigDot`) object, Pacman increases its score by 100 points (resp. 300 points) while the `smallDot` (resp. `bigDot`) object disappears, as a result of the collision. Also, rule *eatCherry* (see Lines 60–63) deals with the situation when Pacman eats (i.e., collides with) a piece of fruit (i.e., a cherry). In this case, the Pacman evolves to be invincible and a transition to the state *pacmanPowerUp* (declared in Lines 70–90; game states and state transitions are described below) is executed (in Line 61) by assigning the value `game-state-transition` (see rule 23 in Table 2) to the result of the action affecting the Pacman. The last rule *ghostCatchPacman* (Lines 64–67) defines the situation in which the Pacman and a ghost coincide in a cell of the map (i.e., they collide) with the consequence that Pacman dies. The consequence is that Pacman teletransports to a random position (see rule 13 in Table 2) whereas the ghost does nothing in particular.

9) END CONDITIONS

This component is used to specify when the game ends according to several different aspects of the game that can be configured (game timeout, number of turns, zero lives event, etc.). Note that a game rule can also be configured as an end condition as seen before and forces the game to finish if it is applied (see rule action `end-game` in Line 21 of Table 2).

The syntax employed to declare end conditions in the XVGDL is:

```
<endConditions>
  <endCondition checkerClass=... attributes />
  <endCondition checkerClass=... attributes />
  <endCondition .../>
</endConditions>
```

A number of distinct criteria to finish the game can be declared with `<endCondition ... />`. The `checkerClass` attribute can be used to specify an external component that we make responsible for evaluating the ending condition. In addition, each ending condition can have a number of attributes that are basically parameters for the external checker in order to check whether the state to establish the end of the game has been reached. In the current version of XVGDL attributes like `value`, `property`, or `objectNames` can be used, just to mention some of them.

As an example, see Lines 70–76 in Appendix, where three conditions to end the game have been declared: after a timeout without user interaction (Lines 71–72), when there are no more volatile objects in the map (i.e., Pacman has collected all the dots in the map, considered as a winning condition; Lines 73–74), or when Pacman loses all its lives (considered as a no-winning condition; Line 75).

10) GAME STATES

In addition to the aforementioned components, XVGDL also provides the element `gameStates` to allow the definition of specific game states during gameplay. The syntax of this component is as follows:

```
<gameStates>
  <gameState id=STR>
    <rules>...</rules>
    <events>...<events>
    <!-- Other game components -->
    ...
  </gameState>
  <gameState id=STR>
    ...
  </gameState>
  ...
</gameStates>
```

It is possible to declare distinct game states. A game state is composed by a name (i.e., a text string STR), a set of game rules and a set of events. In addition, other possible components, such as end conditions, can be associated with a game state.

In an XVGDL game specification, all the elements declared outside the *gameStates* part represent elements that should be applied, affected or considered during all gameplay in any state of the game. The elements declared inside a concrete *gameState* tag will only be considered when the gameplay can be associated to that specific state (as a result of a state transition).

So, we can define specific states grouping elements that are applied (or executed) not only during all gameplay but also under certain circumstances. This is a powerful mechanism that enables the execution of different behaviours during different game phases or after a concrete moment, for instance after an event occurs.

All rules, events and end conditions defined in the default game state (i.e., the state declared outside the *<gameStates>* declaration) will be applied in other states but can be overridden in a concrete game state.

As an example of use, see Lines 78–90 in Appendix. Here we define a specific state, named as `pacmanPowerUp`, associated with the ingestion of a power-up by the Pacman. The consequence is that, at that moment, the Pacman should evolve to beat the ghosts. To cover this situation, this state redefines – in Lines 81–84 – the game rule *ghostCatchPacman* whose definition associated with the default game state was declared in Lines 64–66. The gameplay will transit to this new state when the Pacman eats a cherry, an action that is reflected in Lines 60–62, as already explained. The state of the game can be changed again by a specific rule action declared as before.

The gameplay can be set to its default game state by the activation of an event with result `game-state-reset` as shown in Line 888 (see also, Row 24 in Table 2). The result is that the game returns to its original state, 10 seconds after the activation of the event.

IV. A PROTOTYPE OF AN XVGDL INTERPRETER

XVGDL Game Engine is a prototype of an XVGDL interpreter that allows XVGDL game specifications to be executed in two different ways. First, a human can play an XVGDL

game specification, using, for now, the keyboard as the main input. This can be done by using the keys for the classical movement actions (i.e., left, right, up, down, fire, etc.) that have been included by default in the XVSD, that is to say, the specific XML Schema Definition (VSD) for XVGDL. The keys to interact can also be (re)defined in the `control` component of the specifications as aforementioned. Second, XVGDL Game Engine supports executing games in a simulation mode. So, designers or researchers can create games that can be executed with no human intervention. This can be done by letting all the players be controlled by game AIs managed externally (as shown in the last section). This second approach is important from the point of view of research as it allows games to be specified by XVGDL and played automatically.

One of the main features of XVGDL Game Engine is that, as already mentioned, there are some game aspects (e.g., game interface) that can be specified externally from the game specification itself, allowing different versions of the same game to be defined if necessary. In this prototype, for instance, a basic ASCII-based implementation for XVGDL Game Engine has been developed. This version allows games written in XVGDL to be played and visualise its execution in text-mode. At this point of the research, the visualisation of the game execution itself is not really important, so this renderer is suitable for our needs as it demonstrates the capabilities of XVGDL and the XVGDL engine. It also proves that it is possible to create an executable version of a game from its XVGDL specification. For future implementations, and other targets more focused on commercial uses of the game, tools like Java FX, Android, Java Swing, Unity, Unreal or other solutions should be explored to improve the XVGDL Game Engine.

A. XVGDL GAME ENGINE IN ACTION

Note that XVGDL Game Engine (XGE) is a basic and simple game engine built exclusively for research purposes and to demonstrate the XVGDL features put into action. However, the current implementation acts like a real game engine, taking as input the XVGDL specification of a video game. The XVGDL game engine then process this information by loading each component of this specification. So, it loads the parameters of the renderer configuration (e.g., in our current implementation XGE, the size of the window to show text assocciated with the game execution). It also places the objects according to their specified positions and creates a textual version of the map by executing an external file or by loading a predefined map structure, etc. These two tasks are done according to the specifications given in the components `layout` and `map` of the game specification.

The XGE loads the rest of the information provided in the game specification and it initiates the classical iterative process associated with the execution of games. This basically means to apply the mechanics of the game, let the objects/-players act according to their assigned game AI or in response to the interaction with the user, determine if scheduled game

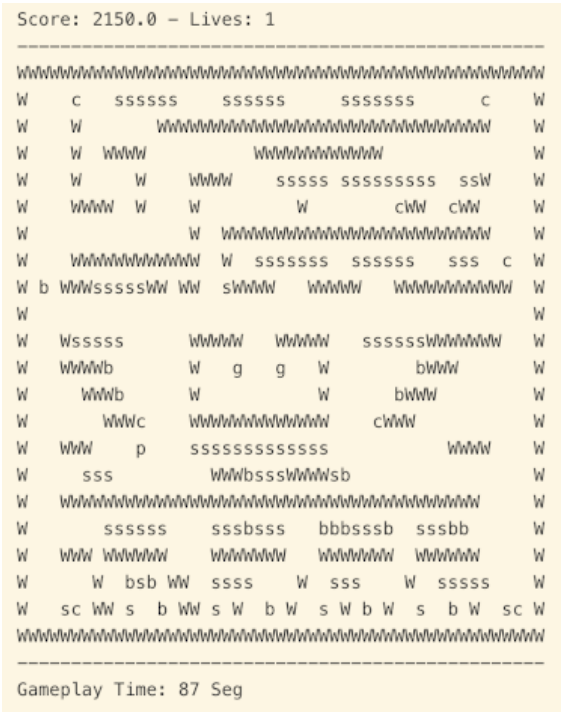


FIGURE 3. Pacman Gameplay. In this capture, w: (W)alls; c: (C)herries; g: (G)hosts; p: (P)layer; b: (B)ig Dots; s: (S)mall Dots.

events and consumable events have to be applied, check the game rules between objects to value if they have to be used (if so, rule actions are applied for each concrete object involved in the rule), and determine whether or not some of the game end conditions have been reached. Then, the information shown in the display window (i.e., a text-window in the current version of XGE) is refreshed and all the (probably modified) information of the game (i.e., the game map, the game objects, etc.) is drawn again. A new iteration is initiated unless an end condition is reached. The game ends when a ending condition has been reached from any of the configured rules.

The prototype of GVE is accessible online and open source can be downloaded from github [23]. XVGDL Game Engine is packaged inside a runnable jar file. XVGDL Games can be launched using this sentence (given <config> is a XVGDL configuration file:

```
java -jar xvgdl-game-engine.jar <config>
```

As an example of execution, Figure 3 shows a screenshot of the XVGDL Pacman specification being executed in XGE. The walls and dots were randomly placed in the map, and later adjusted to have the classical format of the original Pacman game). In this example we used predefined game AIs to govern the behaviour of the ghosts (more specifically, they are trying to catch Pacman). Figure 4 shows a screenshot of the XVGDL Breakout specification running in GVE. The game specification for this well-known game is available at [22]. In this case, the map was generated using an external file-based map generator and all objects were located in the screen according to it.

TABLE 3. XVGDL vs. other existing VGDLs: FEATURES: 'Playable Game' (PF), 'Portable Format' (PF), Standard (STD), Extensibility (EXT). SUPPORT FOR: Multiplayer (MPl), multiple-level games (MLe), Game Rules (GRu), Game Events (GEv), Game Properties (GPr), 2D Games (2D), 3D Games (3D), Artificial Intelligence for objects (AI), independent and separate game renderers (Ren). Symbol ● means that the VGDL referenced in the corresponding row has the feature indicated in the corresponding column whereas symbol ❍ means that the VGDL has this feature but with exceptions (i.e., a limitation to 2 players).

	Features				Support for ...								
	PG	PF	STD	EXT	MPl	MLe	GRu	GEv	GPr	2D	3D	AI	REn
XVGDL	●	●	●	●	●		●	●	●	●	●	●	●
PyVGDL [8]	●	❍	❍	●			●	●	●	●	●		●
TWVGDL [3]				●	●		●	●	●	●			
Ludi [24]	●			●	❍		●	●	●	●			
Ludocore [13]	●			●	●		●	●	●	●			
Casanova [11]	●			●	●	●	●	●	●	●	●		●

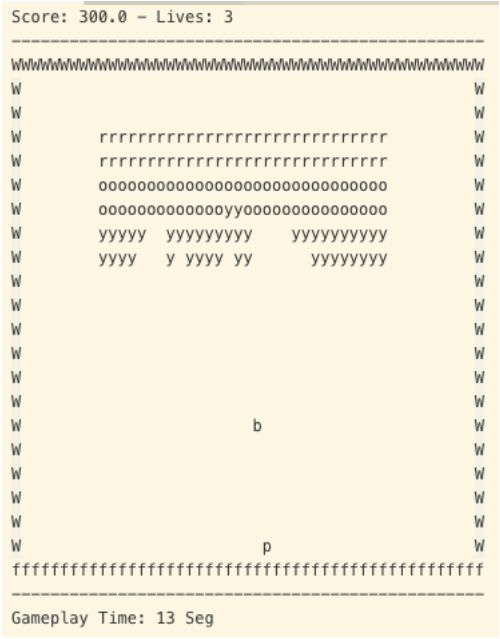


FIGURE 4. Breakout Gameplay. In this capture, w: (W)alls; y: (Y)ellow Bricks; y: (O)range Bricks; y: (R)ed Bricks; p: (P)layer; b: (B)all; f: (F)loor.

V. XVGDL VS. OTHER VGDLS

Now, we compare XVGDL with other interesting VGDLs, most of which have already been mentioned previously (see Section II). We note that none of the existing approaches or implementations for VGDLs have become a standard in the field of video games. In general, most of the existing approaches concentrate on concrete types of games or theoretical aspects. However, XVGDL tries to cover a wide range of games and, powered by its extensibility, could cover other non-supported elements, in the current version of XVGDL, for future needs. In what follows, we highlight the main differences and similarities of our proposal with these.

Table 3 reflects the main features and differences with respect to XVGDL of a number of VGDLs, already reported in the scientific literature. The first part of the table focuses on features that the language provides whereas the second part concentrates on the existence of strictures imposed by the language, to support the definition of specific game components. The first column cites the names (and a corresponding bibliography reference) of the VGDLs included in the comparison. Columns 2 to 5 enumerate distinct features that are desirable in a VGDL. So, given a game specification *GS*, that is to say, a game described in a VGDL, PG ('Playable Game') indicates that this specification can be compiled or interpreted into an executable by some kind of available tool; PF ('Portable Format') means that the GS can be directly managed in a specific way by other available specialized software (e.g., visual editors, validators, or converters to other formats like HTML or Excel, to name a few); STD indicates that the VGDL is based on a well-known programming language (the absence of this feature means that the language has been constructed from scratch with its own notation); and EXT refers to the capacity of the language to be extended (for instance, to support new game components or to cover new genres of games). Columns 6 to 14 refer to the support that the language provides for defining certain game components.

In the case of PyVGDL, the symbol ❍ for the portable format column is used to mark that feature as available, but only under the PyVGDL developing framework. In the case of XVGDL, it is marked as portable because there are hundreds of tools that manage XML files. So, these tools can transform an XML file as needed (for instance, by parsing the XML specification in Python or any other programming language, or by translating the XML file to other file formats, just to give a couple of examples). Thus, XVGDL is not limited to any one development framework or IDE. Likewise, the column standard is marked as ❍ for PyVGDL as it is based on the Python language, a common scripting language nowadays, but not, in fact, a standard. The majority of the GDLs included are not based on any standard language. This means that files written in these VGDLs have to be treated as mere text files. However, files written in XVGDL can be managed by hundreds of software tools dedicated to XML; for instance, (possibly visual) XML editors, translators from XML to other formats (e.g., Excel, Access, HTML, text, etc.), XML content sorters, XML validators, XML debuggers, XML markers, XML viewers, XML schema tools, and many others.

Not only XML-oriented tools are important at this point. It is really interesting that all modern high-level programming languages, such as Java, C or C#, provide support (e.g., in a native way or through specific libraries) to manage XML files. This contributes with much flexibility and allows developers to think about different possible implementations

for XVGDL game engines or interpreters. Developers are not tied to any particular language, rather they are free to use any of them.

Nevertheless, XVGDL supplies most of the evaluated features and multi-level is planned for upcoming versions. Also note that, many of the VGDLs proposed thus far mainly focus on simple 2D graphical games and usually are limited to one discrete state and action space (XVGDL allows, as shown above, the specification of multiple states and state transitions). Moreover, the syntax and semantics of the languages pretty much restrict the specifications to the complexity of primitive board games. This is a consequence of the nature of these VGDLs, limited by their syntax as it was created specifically for particular games. While the XVGDL approach tries to reach the abstraction for a general definition of any kind of game, existing approaches are focused on a particular example (for example, board or arcade games). Rules or mechanics represented in these existing VGDLs are then intended to cover those needs. The exceptions are PyVGDL and XVGDL, although our proposal provides extra features as shown in Table 3.

In addition, as already mentioned, no existing VGDL can be considered a standard for GVGP. So, many proposals for VGDL offer simple structures, with very specific notations, that allow specifying very simple games. Our proposal is more general as it is based on a well-known language, the XML, and takes advantages from it. However, note that XVGDL cannot be considered a standard for VGDLs yet. Moreover, while most of the work done with VGDLs has been directed to concrete examples (particular development for a known game), XVGDL tries to change the approach, focusing first on the abstraction of defining games and thereby letting us create specifications for a wide range of games.

VI. CONCLUSION AND FUTURE RESEARCH

This paper has presented XVGDL, a video game description language which is based on XML (a well-known markup language) and can be used to write video game specifications. As any video game description language, it has structures to specify video game components such as game layout, game mechanics, or game objects. In addition, XVGDL covers other many important game components such as multimedia elements, game artificial intelligence, (procedural) map generators or game states. Moreover, XVGDL also provides additional advantages that no other VGDL has. Note that XVGDL is based on XML so that the descriptions of games are edited as XML files, following the XML syntax with its tags, properties and components. This means that XVGDL game specifications can be managed by any software able to manage XML files, including generic XML tools.

Designed to be a video game description language, our proposal also eases the transition from game specifications to game implementations and provides structures to deal with this issue. For instance it allows specifying the data input for the interaction with the user(s), or conceding the execution of specific game tasks to external files (e.g., for the procedural generation of maps, for checking ending conditions, or for controlling the behaviour of non-player characters). In order to validate our ideas, in this paper we have also presented a first prototype implementation for the XVGDL Game Engine (XGE), a tool that enables an executable version of a game to be obtained directly from it specification written in XVGDL.

The XVGDL language is open to incremental modifications to meet any further needs. New game-related concepts, not included in the current version of XVGDL presented here, can be tackled. Note that we have defined an XML Schema Definition for XVGDL. This schema can be employed for validating our game specifications written in XVGDL. New language artifacts can be offered to the game designers by extending the schema adequately. For the future, we are considering including multiple level management and support for narrative definition.

Another line of future research is to explore how XVGDL can be extended to deal with games that manage imperfect information such as the digital collectible card games (e.g., HearthStone) or poker. Finally, we plan to improve the prototype implementation of XVGDL Game Engine.

In accordance with the primary principles of Open Science, the sources of the XVGDL game Engine (XGE) and the XML Schema Definition for XVGDL are publicly available (see [4], [23]). In addition, many details of XVGDL and a number of game specifications written in XVGDL can be found at [22].

APPENDIX
PACMAN XVGDL EXAMPLE

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
2  <!-- Definition~of pacman game -->
3
4  <gameDefinition>
5    <!-- Renderer configuration -->
6    <property key="rendererConfiguration" value="/context/pacmanAsciiRendererConfiguration.xml" />
7    <!-- Timeout configuration in milliseconds. Set to -1 for no \timeout -->
8    <property key="timeout" value="20000" />
9
10   <layout>
11     <component id="gameInfoTop" location="top">
12       <contextProperty id="score" />
13       <contextProperty id="hiScore" />
14     </component>
15
```

```
  <component id="gameInfoDown" location="bottom">
    <contextProperty id="lives" />
    <contextProperty id="playerItems" />
  </component>
</layout>

<!-- XVGDL allows define controls that can be overriden whithin controls tag
    In case of Pacman, no~fire, jump or special control is defined
-->
<controls>
  <control left="27" right="26" up="24" down="25/>
</controls>

<map type="2D" sizeX="20" sizeY="40" toroidal="false"
  generator="es.jor.phd.xvgdl.model.map.RandomLocationGameMapGenerator">
</map>

<players number="1" maxNumber="1" minNumber="1">
  <player name="pacman" score="0" lives="3" livePercentage="100"/>
</players>

<objects>
  <object name="wall" type="wall" dynamic="false" volatile="false" sizeX="1" sizeY="1" instances="60" />
  <object name="ghost" type="enemy" dynamic="true" volatile="true" size="1,1" instances="2"
     ai="chase_player" />
  <object name="ghost" type="enemy" dynamic="true" volatile="true" size="1,1" instances="2"
     ai="random_movement" />
  <object name="smallDot" type="item" dynamic="true" volatile="true" size="1,1" instances="2" />
  <object name="bigDot" type="item" dynamic="true" volatile="true" size="1,1" instances="2" />
  <object name="cherry" type="item" dynamic="true" volatile="true" size="1,1" instances="1" />
</objects>

<events>
  <event type="engine" className="es.jor.phd.xvgdl.model.event.SpawnItemEvent" objectName="cherry"
     timer="5000" />
  <event type="engine" className="es.jor.phd.xvgdl.model.event.SpawnItemEvent" objectName="ghost"
     timer="1000" />
</events>

<rules>
  <rule name="eatSmallDot" type="collision">
    <ruleAction objectName="pacman" result="score-up" value="100" />
    <ruleAction objectName="smallDot" result="disappear" />
  </rule>
  <rule name="eatBigDot" type="collision">
    <ruleAction objectName="pacman" result="score-up" value="300" />
    <ruleAction objectName="bigDot" result="disappear" />
  </rule>
  <rule name="eatCherry" type="collision">
    <ruleAction objectName="pacman" result="game-state-transition" value="pacmanPowerUp" />
    <ruleAction objectName="cherry" result="disappear" />
  </rule>
  <rule name="ghostCatchPacman" type="collision">
    <ruleAction objectName="pacman" result="teletransport" />
    <ruleAction objectName="ghost" result="" />
  </rule>
</rules>

<endConditions>
  <endCondition checkerClass="es.jor.phd.xvgdl.model.endcondition.TimeoutGameEndCondition"
    property="timeout" value="0" />
  <endCondition checkerClass="es.jor.phd.xvgdl.model.endcondition.NoObjectsPresentGameEndCondition"
    objectNames="bigDot,smallDot" winningCondition="true"/>
  <endCondition checkerClass="es.jor.phd.xvgdl.model.endcondition.LivesZeroGameEndCondition" />
</endConditions>

<gameState id="pacmanPowerUp">
  <rules>
     <!-- Overrides rule for ghostCachPacman -->
     <rule name="ghostCatchPacman" type="collision">
       <ruleAction objectName="pacman" result="score-up" value="500" />
       <ruleAction objectName="ghost" result="disappear" />
     </rule>
  </rules>
```

```
    <events>
      <!-- Back to normal state after a configured time
      <event type="engine" result="game-state-reset" timer="10000" />
    </events>
  </gameState>
</gameDefinition>
```

REFERENCES

[1] J. Levine, C. B. Congdon, M. Ebner, G. Kendall, S. M. Lucas, R. Miikkulainen, T. Schaul, and T. Thompson, "General video game playing," in *Artificial and Computational Intelligence in Games* (Dagstuhl Follow-Ups), vol. 6, S. M. Lucas, M. Mateas, M. Preuss, P. Spronck, and J. Togelius, Eds. Dagstuhl, Germany: Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, 2013, pp. 77–83, doi: 10.4230/DFU.Vol6.12191.77.

[2] G. N. Yannakakis and J. Togelius, "A panorama of artificial and computational intelligence in games," *IEEE Trans. Comput. Intell. AI in Games*, vol. 7, no. 4, pp. 317–335, Dec. 2015, doi: 10.1109/tciaig.2014.2339221.

[3] M. Ebner, J. Levine, S. M. Lucas, T. Schaul, T. Thompson, and J. Togelius, "Towards a video game description language," in *Artificial and Computational Intelligence in Games* (Dagstuhl Follow-Ups), vol. 6, S. M. Lucas, M. Mateas, M. Preuss, P. Spronck, and J. Togelius, Eds. Dagstuhl, Germany: Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, 2013, pp. 85–100, doi: 10.4230/DFU.Vol6.12191.85.

[4] J. Ruiz and A. J. Fernández-Leiva. (2018). *XVGDL Schema*. [Online]. Available: https://github.com/jorgeruizqui/phd/blob/master/xvgdl/xvgdl-core/src/main/resources/xvgdl.xsd

[5] G. Jiang, D. Zhang, L. Perrussel, and H. Zhang, "Epistemic GDL: A logic for representing and reasoning about imperfect information games," in *Proc. 25th Int. Joint Conf. Artif. Intell. (IJCAI)*. New York, NY, USA: AAAI Press, 2016, pp. 1138–1144. [Online]. Available: http://dl.acm.org/citation.cfm?id=3060621.3060779

[6] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth, "General game playing: Game description language specification," Stanford Univ., Stanford, CA, USA, Tech. Rep. LG-2006-01, 2008.

[7] M. Thielscher, "A general game description language for incomplete information games," in *Proc. 24th AAAI Conf. Artif. Intell. (AAAI)*, Atlanta, GA, USA, M. Fox and D. Poole, Eds. New York, NY, USA: AAAI Press, Jul. 2010. [Online]. Available: http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1727

[8] T. Schaul, "An extensible description language for video games," *IEEE Trans. Comput. Intell. AI in Games*, vol. 6, no. 4, pp. 325–331, Dec. 2014.

[9] *Unity Game Development Platform*. Accessed: Nov. 15, 2019. [Online]. Available: https://unity3d.com/es

[10] Casanova. (2012). *Casanova*. [Online]. Available: https://github.com/vs-team/casanova-mk2

[11] M. Abbadi, F. Di Giacomo, A. Cortesi, P. Spronck, G. Costantini, and G. Maggiore, "Casanova: A simple, high-performance language for game development," in *Serious Games*. Cham, Switzerland: Springer, 2015, pp. 123–134.

[12] C. Browne, "Evolutionary game design: Automated game design comes of age," *SIGEVOlution*, vol. 6, no. 2, pp. 3–16, Feb. 2014, doi: 10.1145/2597453.2597454.

[13] A. M. Smith, M. J. Nelson, and M. Mateas, "LUDOCORE: A logical game engine for modeling videogames," in *Proc. IEEE Conf. Comput. Intell. Games*, Aug. 2010, pp. 91–98.

[14] T. Schaul, "A video game description language for model-based or interactive learning," in *Proc. IEEE Conf. Comput. Intell. Games (CIG)*, Aug. 2013, pp. 1–8.

[15] *World Wide Web Consortium*. Accessed: Nov. 15, 2019. [Online]. Available: https://www.w3.org/xml/

[16] *Pacman Official*. Accessed: Dec. 2, 2019. [Online]. Available: http://pacman.com/en/

[17] M. Wickramasinghe, K. Gunawardana, J. Rajapakse, and D. Alahakoon, "Investigating individual game-play patterns using a self-organzing map," in *Proc. IEEE 6th Int. Conf. Inf. Autom. Sustainability*, Sep. 2012, pp. 203–208.

[18] A. Chiang, "Motivate AI class with interactive computer game," in *Proc. 1st IEEE Int. Workshop Digit. Game Intell. Toy Enhanced Learn. (DIGITEL)*, Mar. 2007, pp. 109–113.

[19] J. Svensson and S. J. Johansson, "Influence Map-based controllers for Ms. PacMan and the ghosts," in *Proc. IEEE Conf. Comput. Intell. Games (CIG)*, Sep. 2012, pp. 257–264.

[20] Q. Sun and S. He, "Artificial neural network using the training set of DTS for Pacman game," in *Proc. 11th Int. Comput. Conf. Wavelet Actiev Media Technol. Inf. Process. (ICCWAMTIP)*, Dec. 2014, pp. 209–213.

[21] J. Ruiz and A. J. Fernández-Leiva. (2018). *XVGDL Configuration for Pacman*. [Online]. Available: https://github.com/jorgeruizqui/phd/blob/master/xvgdl/xvgdl-pacman/src/main/resources/context/pacman_context_configuration.xml

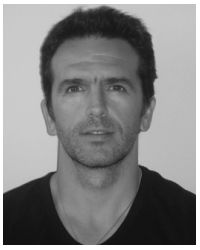
[22] (2019). *XVGDL Web*. [Online]. Available: https://www.xvgdl.com

[23] (2018). *XVGDL Game Engine GitHub Repository*. [Online]. Available: https://github.com/jorgeruizqui/phd/blob/master/xvgdl/build

[24] C. Browne and F. Maire, "Evolutionary game design," *IEEE Trans. Comput. Intell. AI in Games*, vol. 2, no. 1, pp. 1–16, Mar. 2010.

[25] S. M. Lucas, M. Mateas, M. Preuss, P. Spronck, and J. Togelius, Eds., *Artificial and Computational Intelligence in Games* (Dagstuhl Follow-Ups). Dagstuhl, Germany: Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, 2013, vol. 6. [Online]. Available: http://www.dagstuhl.de/dagpub/978-3-939897-62-0

JORGE R. QUIÑONES received the degree in computer science from the University of Granada (UGR), Spain, in 2002. He is currently pursuing the Ph.D. degree in computer science with the University of Málaga (UMA). His entire professional career has been developed as a Software Engineer in private international companies, such as CapGemini or Indra and, more recently, as a Technical Lead at Piksel. Working in different projects and business inside the IT market, he has specialized during almost eight years in air traffic control supervision software. He is also leading cutting-edge technology solutions for the media industry's biggest companies around the world. He has combined his professional career with an extra formation in industrial organisation at UMA.



ANTONIO J. FERNÁNDEZ-LEIVA received the Ph.D. degree in computer science from the University of Málaga (UMA), Spain, in 2002. He worked in private companies as a Computer Engineer. He is currently an Associate Professor with the Lenguajes y Ciencias de la Computación Department. He leads a master studies on game development at UMA, is the Co-Head of the CAESIUM research group at UMA, and is also the Co-Founder of A Bonfire of Souls, a private game development company. His main areas of research involve both the application of metaheuristics techniques to combinatorial optimization and the employment of computational intelligence in games.