Model-Driven Serious Game Development: Integration of the Gamification Modeling Language GaML with Unity

Amir Matallaoui
Technical University of Berlin
Straße des 17. Juni 135
Berlin, Germany
amir.m@ikm.tu-berlin.de

Philipp Herzig
SAP SE
Konrad Zuse Ring 10
Potsdam, Germany
philipp.herzig@sap.com

Rüdiger Zarnekow
Technical University of Berlin
Straße des 17. Juni 135
Berlin, Germany
ruediger.zarnekow@ikm.tu-berlin.de

Abstract

The development of gamification within non-game information systems as well as serious games has recently gained an important role in a variety of business fields due to promising behavioral or psychological improvements. However, industries still struggle with the high efforts of implementing gameful affordances in non-game systems. In order to decrease factors such as project costs, development cycles, and resource consumption as well as to improve the quality of products, the gamification modeling language has been proposed in prior research. However, the language is on a descriptive level only, i.e., cannot be used to automatically generate executable software artifacts. Therefore, domain experts are able to understand and modify existing formal gamification definitions.

In this paper and based on this language, we introduce a model-driven architecture for designing as well as generating building blocks for serious games. Furthermore, we give a validation of our approach by going through the different steps of designing an achievement system in the context of an existing serious game.

1. Introduction

Gamification, defined as the use of game design elements in non-game context [1], is an interdisciplinary approach, through which users are motivated to achieve certain behavioral or psychological outcomes (e.g., learn faster, complete profile, use specific platform daily). However, in industry, the majority of gamification concepts are never implemented after their conceptualization. One key issue is that the effort for the subsequent implementation of the concept requires high effort with regards to time and development resources on the one hand and to the fact that the desired outcomes of the gamification cannot be guaranteed on the other hand.

Therefore, it is necessary to simplify the overall implementation process of gamification. In order to improve the communication between domain and gamification experts as well as IT experts, the Gamification Modeling Language (GaML) has been proposed [2]. GaML defines a meta-model of game design elements based on the current consensus of game design elements in the gamification community. Besides its formal definition it has been shown that the language is highly understandable by both target groups [3]. Therefore, domain experts are able to understand and modify existing formal gamification definitions.

So far the GaML is of descriptive nature only, i.e., it is possible to check whether a given specification is valid with regards to the meta-model and additional semantic properties. However, a valid specification cannot be used to automatically generate software artifacts that can be utilized to assemble a gamified system or parts thereof. Therefore, we aim at the generation of target code from the corresponding models by means of a given model-to-code transformation, i.e. models in GaML should be deployed to substitute code and to serve as input for code generators.

Through integrating GaML in the automated process of creating gamification and serious games, we argue that the efforts for implementing gamification artifacts can be significantly reduced. Furthermore, the language introduces a higher level of abstraction since the approach allows being independent from concrete technologies, platforms, or specific computational models. Furthermore, the approach helps to increase the overall quality of the gamified software since errors in the implementation can be already detected and fixed at design time which is not the case in current embodiments.

These facts, however, have not been proven and there is no published work showing a real case implementation of GaML. In this paper we present our work, in which we have gone through the process of generating target code from the model and integrating it into a target application. We thereby argue that GaML can also be used in the implementation of serious games.

The paper is structured as follows. First, we present a model-driven approach for defining gamification and building blocks of serious games. We thereby also illustrate the importance of adopting such an ap-
proach. Second, we present a model-driven architecture for developing achievement systems within serious games made in Unity. Third, we validate the given architecture by presenting a case study. The paper closes with a summary and an outlook to future work.

2. Model-driven serious game development

Model-driven software development (MDSD) is defined in [4] as the deployment of domain-specific languages (DSLs) in order to create models that efficiently express the structure as well as the behavior of an application. These models are afterwards transformed into compilable code by means of predefined model-to-code transformations.

MDSD features various advantages if compared to conventional software development. On the one hand, it improves the integration of professional domain experts in the implementation of the application. This enhances the development process by increasing the communication between designers and developers. On the other hand, a better quality control of the application’s behaviors and rules is achieved by the designers themselves, since they are able to read and understand the language elements, as they were an active part during the development. Moreover, domain-experts are able to program in the DSL themselves, and may directly define the technical behavior of the application.

Being a special case of software, serious games development should also profit from the above stated advantages (see Figure 1). It has already been mentioned that gamification concepts can be expressed using GaML [2]. Taking this in consideration, we next show that GaML can as well be used in developing special building blocks for serious games.

2.1. The gamification modeling language

The Gamification Modeling Language has been initially proposed as a formal language adhering to a context-free grammar and is based on the current consensus of game design elements that can be found in the gamification literature (e.g., [5]) and is consistent with prior models and taxonomies (e.g., [1]). Besides the formal definition of gamification concepts, it has been shown that the language’s meta-model, i.e., the abstract syntax, is understandable and modifiable for domain experts, i.e., when they are confronted with a given instance [3]. In addition, GaML has two concrete syntaxes, namely a textual and a graphical form-based one. It has been shown that understandability and modifiability is independent of the representation, however, the graphical is better suited when writing new instances [6]. For the remaining text we use the textual syntax for a compact visualization.

Figure 1: Current vs. model-driven gamification development approach [2]

By using GaML we ensure one of the core features of MDA, which is portability. This means that models are designed platform-independently since they only describe abstract concepts and are not related to any system’s architecture. Further advantages of integrating GaML within serious games development include:

First, GaML models are expressed using textual input and this makes it easy to learn; by using text, we can describe any technical details of the developed application. Second, there exist diverse tools which support text input and textual artifacts. This can be very crucial, especially when considering the model’s life cycle and its versioning. Third, gamification as well as serious game concepts could be defined by designers who do not need to know how application

---

1 The grammar in Extended Backus-Naur Form can be found under: https://github.com/AmirIKM/GamiAsService/blob/master/GaMLGrammar.xtext
development is done using specific development environments. This ensures a clear task separation. Furthermore, the consistency of the development is guaranteed, since designing tasks are done outside the used application development IDE. GaML, the language used for the definition of achievements, ensures that all approved instances conform to the language’s grammar and that these well-formed instances should compile to the target language [2].

3. Model-driven achievement system design for Unity-made serious games

3.1. Achievement system definition workflow

In this section, we show the model-driven process for creating an achievement system for serious games made in Unity (see Figure 2).

![Figure 2: Achievement system model-driven implementation](image)

Foremost and according to the GaML grammar, achievement designers write an instance of the GaML language, in which they define the desired achievements. Next, the achievements’ definitions are checked for errors and a corresponding target code (in our case a JSON representation) is generated by means of the provided model to code transformation defined using Xtend. After building and running the game, the JSON file is parsed by the Unity achievement system plugin and achievements are then listed in the appropriate interface defined by the game designer.

The above-mentioned steps are described in more detail in the following text.

3.2. Achievement definition in GaML

At first place, a GaML instance is created, in which achievements are defined. As GaML was developed using Xtext, the definition of achievements is text based and is carried out within the Eclipse IDE.

![Figure 3: Xtext Framework](image)

Xtext [7] is an Eclipse plug-in that supports domain-specific language development. Due to this fact, many of Xtext functions are based on already established frameworks in Eclipse and on the features of Eclipse itself. The final outcome of a DSL development in Xtext is again available as an Eclipse plug-in. Hence, it does not only provide a compiler and a code generator but a complete development environment. Furthermore, this can be adjusted to any specific requirements of a particular DSL. The advantage here is that many developers are familiar with such a common environment like Eclipse. A possible disadvantage, however, could be that this tool is directed to a target group of programmers due to its considerable level of complexity. Another special feature of Xtext is the use...
of a text editor as an input medium. Hence, the DSL code (GaML code) is entered by the end-users as free text. They are thereby assisted by automatic code-completion of the inputs, syntax highlighting and other features (Figure 3). A validation of their inputs is as well made using various criteria, which can be defined by the end-users themselves.

However, due to the entering of the GaML code in the form of a free text, Xtext should at first convert the user's input into a semantic model. Afterwards, validations and transformations (e.g., target code generation) are carried out based on the semantic model.

![Figure 4: From code to semantic model](image)

This conversion is performed with the help of a lexer (lexical analyzer) and a parser (Figure 4). In the first instance and according to specific rules, the entered free text is decomposed by the lexer in individual related components; so-called tokens. Based on the lexer’s outcome, the parser interprets the user’s input and translates it into a semantic model (Abstract Syntax Tree) using the defined parser rules. For the projection of this model, the modeling constructs of the Eclipse Modeling Framework are used. Only when this transformation of the free text into a semantic model can be successfully finished, other transformations such as target code generation could be made.

### 3.3. Model to code transformation

The code generators were written using the Xtend language, which was specifically configured to enable programming based on text templates. The information provided by the semantic model is accessible within the Xtend program by using the Java classes, which represent the language’s AST model. These classes are generated automatically via the Eclipse Modeling Framework.

Xtend [8] is a statically typed programming language whose typing, however, is mostly done by type-inference (i.e., automatic deduction of the type of expressions). Therefore, types are rarely explicitly declared. The Xtend language was specifically designed to improve some of the existing Java concepts. Upon execution, Xtend is not directly translated into bytecode but Java code. This makes Xtend programs deployable on arbitrary platforms. Moreover, this has the advantage that the programmer can trace the Java programming constructs, which result from his code. Within Xtext, Xtend is used as a code generator for the implemented DSLs.

The Xtend constructs that were especially intended to be used for code generation include dispatch functions, which are featured by Xtend, in addition to normal functions, whose use is done via a static binding through their parameter values. By using the keyword ‘dispatch’, various functions (having the same name) can be combined into a set of dispatch functions. Within these overloaded functions, it is decided through the type(s) of the input parameter(s), which function is called. The code snippet below shows two dispatch functions having the same identifier “compile” but different input parameters (due to demonstration purposes, the snippet shows a possible way to generate Java code from the GaML model; though in this work JSON code is generated). The “compile” function instantiates an integer and sets its value to 0 if the construct is of type Point. If this latter is of type Badge then a string with the badge’s name is created.

```java
def dispatch compile (Point point) {
    public int «point.name» = 0;
}

def dispatch compile (Badge badge) {
    public string «badge.name» = "";
}
```

One of the main tasks of the code generation is to put together long strings. In order to make this task as simple as possible and obtain a readable code as well, Xtend offers ‘rich strings’. To mark the use of such
3.4. Code deployment

Depending on the target platform, the code that has been generated using the model to code transformation is then integrated in the application. This automatically generated code should not be modified. For instance, it would be inconvenient if after writing C and compiling it we needed to edit and revise the assembly language or machine code produced [9]. Similarly, the generated code form the GaML instance should not be modified or reworked.

4. Case study

In this section we present a real case study, in which we have defined achievements for an existing Unity application. For demonstration purposes, we use a “Stop Smoking” app. This latter is a Unity serious game that aims to help smokers quit smoking by offering simple and easy steps to follow. In order to enhance the motivation we have designed an achievement system, through which players are rewarded badges for accomplishing given tasks.

The described workflow of Figure 2 has been followed for defining the achievements.

4.1. Unity

Unity is a cross-platform game engine with a built-in IDE developed by Unity Technologies. It is used to develop video games for web plugins, desktop platforms, consoles and mobile devices [10].

The Unity game engine is also often used for making serious games. These latter take advantage of Unity’s core features such as physics, networking, cross-compiling and other quite important features. The categories, under which these serious games generally appear, include [11]:

- Education and instruction (e.g., Micro Plant, Surgical Anatomy of the Liver)
- Product demonstration and configuration (e.g., Bike configurator, Gun Construction)
- Architectural visualization (e.g., Insight 3D, Aura Tower)
- Exercise simulations (e.g., US Navy Virtual Training, I.R.I.S.)

4.2. Achievement systems

Achievement systems can be seen as “meta-tasks” (tasks over key-task), which provide further goals to the system users and are independent of the actual main goals. Hamari et al. define achievements as follows: “Achievements are goals in an achievement/reward system (different system than the core game) whose fulfilment is defined through activities and events in other systems (commonly in the core game)” [12].

The below class diagram (Figure 5) shows an adapted interpretation of a possible achievement’s structure presented in [12]. Based on this diagram we can see that achievements consist mainly of three important parts, namely:

- An identifier, which to its turn consists of a name, through which the achievement is made unique, a description (hint) of the logic behind it and a badge, which encompasses its visualization.
- An achievement unlocking-logic, which entails:
  - A trigger: Could be either an action done by the user (e.g., eat an apple) or an external event (e.g., temperature rises by x degrees).
  - Conditions: Arbitrary other conditions under which the trigger is valid (e.g., a user has already another badge assigned).
  - Count: The number how often the action or the event has to be triggered.
  - Pre-requirements: are global requirements, which have generally nothing to do with the conditions mentioned above (e.g., complexity mode should be set to hard in order to achieve this task).
- A Reward, by which users are compensated for unlocking the achievement. This reward can whether be game-related (e.g., points), system-related (e.g., by unlocking achievement X you fulfill on of the condition for unlocking achievement Y) or application-external (e.g., users are rewarded a shopping coupon).
4.3. Unity achievement system plugin

The Unity plugin offers a generic achievement system that can simply be integrated in the application by importing it into the project folder. At this step, if we run the application just after integrating the plugin and without having defined achievements in GaML, the Unity inspector window shows an empty achievements list.

The screenshot in Figure 6 shows properties of such a list in a form-based way. This form is later filled in with properties that were parsed out the JSON file. These properties include the achievement’s name, description, badge value, badge icon, target count, current count and if the achievement has already been earned or not.

4.4. Achievements definition in GaML

In the following text, an excerpt from the GaML specification of our use case is shown. The instance is introduced with the concept keyword followed by the concept’s name, here NonSmoking. Furthermore, the concept defines one user action, namely the action that is triggered by the user if he or she hasn’t smoked for a certain amount of days.

In addition, the concept defines two point metrics. First, RewardPoints that are assigned upon completion of specific missions and are visible to the user all the time. It is important to note that the point is declared of type Advancing which signified that these points cannot be lost. Second, NonSmoking-DaysPoints which are used to track internally how many days the user has not smoked. Consequently, the point is of type Auxiliary which is used for the code generation at a later point in time. This point is not explicitly shown to the user but is used for tracking purposes only (e.g., to show them in a progress bar). This second point definition also covers a game rule that says every time users trigger the did_not_smoke event they get exactly the numbers of NonSmokingDayPoints as specified in the day attribute of the user action.

Besides the point definitions, the specification also defines a badge concept as meta-data. In accor-
dance with the achievement model provided before, the badge contains a name, description and a link to an icon as defining characteristics.

```concept NonSmoking {
    user action did_not_smoke {
        properties {
            day : Number
        }
    }
}
```

```point RewardPoints {
    name = "Reward Points",
    abbreviation = "RP",
    type = ADVANCING
}
```

```point NonSmokingDayPoints {
    name = "NonSmokeDay",
    abbreviation = "NSD",
    type = AUXILIARY
    when player {
        did u1 : user action did_not_smoke
        then {
            give u1.day NonSmokingDayPoints
        }
    }
}
```

```badge One_Week {
    name = "1st Week",
    description = "1 whole week without Smoking!",
    image = ".//images/7free.png"
}
```

```mission OneWeek {
    name = "One Week Smoking",
    description = "Mission is completed when player spends 7 days without smoking."
    when player {
        has point NonSmokingDayPoints, SUM=7
        then {
            give badge First_Week,
            give 10 RewardPoints
        }
    }
}
```

Finally, the concept covers the notion of missions which represents explicit building blocks to define condition-action pairs based on the user interaction or state. In this simple example, the mission requires the user to have seven `NonSmokingDayPoint`. If collected, the mission assigns the previously defined badge as well as 10 additional `RewardPoints` to the user

4.5. Model to code transformation

In order to generate code from our GaML model, we have defined a GaML to JSON transformation using the Xtend language. The Xtend code below shows the main part of the GaML-to-JSON transformation. Rich strings have been used to ensure the target format.

```«FOR b:nonSmoking»
    '''
    {
        "Name": «b.getName»,
        "Description": «b.getDescription»
    }
    "Badge": {
        "Value": «b.getMission.getReward»,
        "Icon": «b.getIcon»
    },
    "Count": {
        "target": «b.getMission.when»,
        "current": 0,
        "earned": false
    },
```

The code snippet below shows the corresponding fragment of the generated Json code.

```{
    "Name": "1st week",
    "Description": "1 whole week without Smoking!",
    "Badge": {
        "Value": 10,
        "Icon": ".//images/7free.png",
    },
    "Count": {
        "target": 7,
        "current": 0,
        "earned": false
    }
```

4.6. Run application

Before running the application, we have to make sure that we have imported the Unity Achievement System plugin into our project. Then, when we run the application, the JSON code is parsed by the plugin and the inspector form is automatically filled in with the wished achievements as well as their corresponding properties.

Please note that this is an oversimplified example for demonstration purposes only. In order to correctly model temporal constraints such as one week, the definition has to be defined using time windows as explained in the official GaML documentation.
The screenshot in Figure 7 shows the application’s achievement controller, which consists of an array of the defined achievements. The array contains two achievements (Size = 2). The first one has the identifier “1st day”, is rewarded 2 points (badge value = 2) and is earned when the “player” stays one day (target count = 1) without smoking. The second one has the identifier “1st week”, is rewarded 10 points (badge value = 10) and is earned when the “player” stays 7 days (target count = 7) without smoking. Both achievements are not earned yet.

Figure 8 shows the graphical representation of the achievement with the identifier “1st week”. Most of the achievement properties can be seen except its icon. This latter is hidden by the description, which pops up on mouse hover.

The plugin offers also a locked version as well as an unlocked one for each badge (see Figure 9).

5. Related works

The state of the art in the field of model-driven development of gamification as well as serious games lacks a concrete approach that shows a complete MDA going through the whole framework and taking advantage of domain-specific languages.

In [13] a Serious Game Structure and Logic Modeling Language (GLiSMo) is introduced. The authors state that different serious game components could be modeled using GLiSMo. These include amongst others acts and scenes, objects, characters and inventories, user feedback and rewards. To demonstrate the use of the language, Shack City was designed. It represents a story based serious game, which playfully help learn about sanitation, heating and cooling. However, the game has been developed and implemented in a first version as a web application using HTML5. Thus, GLiSMo was only used for the sake of description and no automatic code generation was targeted.

In [14] the game description language GDL is introduced. This language, although very expressive it is only used to describe and reason about the rules of general games with no intention of serving the further implementation of these rules by offering a model-to-code transformation.

In [15] a visual domain-specific language named SharpLudus (SLGML) is presented. Although encompassing semantic validators and code generators in addition to the language, the proposed approach can be seen more like a game engine than a model-driven way for game development. Authors state also that some classes (e.g., CustomTriggers and Custo-
mReactions) are not re-generated and specific methods should be added by the developers. This violates one of the most MDSD rules, namely generated code should not be modified. Moreover, the presented language is not suited for the development of serious games since, for example, no achievements, rewards or ranking systems can be modeled using it.

Another domain-specific visual language is introduced in [16]. The DSVL makes it possible for game designer to design and run their own games by offering automatically compilable graphical game elements. Although easy to understand and to use, the games created by arranging these graphical elements are restricted to narrative educational games, which deploy exclusively the point-&-click genre.

In [17] authors present a prototype of a serious game, whose goal is to instruct people about fire drills. A virtual fire evacuation coach was therefore developed. It was namely made in Unity but no model-driven approach was adopted and no automatic code generation was conducted.

In [18] authors describe the process of creating a domain-specific modeling language for serious games. They, however, suggest a model-driven framework for designing such games based on UML, which is in fact a general-purpose modeling language. This presented approach loses out on the benefits of domain-specific languages, of which model-driven serious game development should make use as demonstrated in our work.

6. Summary & Outlook

In this paper we have introduced a model-driven approach for developing serious games. We have thereby taken advantage of the gamification modeling language GaML and hence have presented a model-driven architecture for the development of achievement systems within serious games made in Unity. As a validation of the proposed architecture, a case study has been presented.

Next, we are planning to conduct a quantitative evaluation of the proposed approach in order to demonstrate the potential and effectiveness of adopting such an approach in comparison to the conventional development methods.

7. Acknowledgment

This research has been funded by the German Ministry for Education and Research (BMBF) under the project number 01IS12056. The responsibility lies with the authors.

8. References


