

UgameFeature: Automatic code generation for Unity game projects

Joshua Kritz^a, Mart de Roos^b, Luís Ferreira Pires^c, João Luiz Rebelo Moreira^d,
Giancarlo Guizzardi^e

Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente, Enschede, the Netherlands
{j.kritz, l.ferreirapires, j.luizrebelomoreira, g.guizzardi}@utwente.nl, m.c.deroos@student.utwente.nl

Keywords: Model-Driven Engineering, automatic code generation, Unity, game software


Abstract: Computer games are complex software systems, which means that their development requires some level of programming skills. However, their design also involves the creation of game objects (characters, scenarios, etc.), animations and story lines, which are designed by game domain experts, who in general have minimal (or no) programming skills. Game engines have been developed to facilitate game development by reducing programming efforts and enhancing productivity, but we observed that most of these engines still require programming skills in order to be used. In this paper, we discuss how Model-Driven Engineering technologies, particularly metamodeling and model transformations, can be used to facilitate game development. We define a Domain Specific Language called UGameFeature to be used by game designers to define games that can be automatically transformed into scripts that can be executed by the Unity game engine. In order to facilitate the code generation step, we defined an intermediate metamodel, so that structural differences between the UGameFeature metamodel and the Unity engine scripts can be accommodated by an intermediate model-to-model transformation. We claim that with this approach we could define a streamlined process to go from game design to game implementation, in this way surpassing the benefits already offered by game engines. We also discuss some practical obstacles of applying MDE techniques and give recommendations to practitioners who want to apply them in their projects.


1 INTRODUCTION


The game industry has been growing in the last years, providing a total revenue of over US\$ 170 billion in 2020 (Wijman, 2021), and attracting a lot of commercial interest. Computer games are complex software systems, which require some level of programming skills in order to be produced. However, their development also involves some creative design in which game objects (characters, scenarios, etc.), animations and story lines are defined. These tasks are often performed by game designers with artistic skills, but minimal or no programming skills. This means that the game design and programming tasks should somehow be separated from each other, similarly to how HTML/CSS and programming languages like JavaScript and Java are used in web applications to separate layout from functionality, respectively.


Games engines have been developed lately to facilitate game development by reducing programming efforts and enhancing productivity. These engines provide most of the code necessary to implement games, and some of them do not require any programming by supporting visual languages to describe game objects and events, like Construct¹ or Stencyl². However, industry still favours more robust and mature engines that demand some programming skills, like Unity³ and Unreal⁴ (Toftedahl, 2021). Unity is the most popular engine amongst novice developers, and therefore in this paper we use it as a game implementation platform.


This paper presents an approach based on Model-Driven Engineering (MDE) technologies to facilitate game development on top of the Unity game engine. Using MDE will provide both a way to abstract from programming language constructs and to generate code (semi-)automatically (Brambilla et al.,

^a  <https://orcid.org/0000-0002-6661-6292>

^b  <https://orcid.org/0000-0001-5551-4488>

^c  <https://orcid.org/0000-0001-7432-7653>

^d  <https://orcid.org/0000-0002-4547-7000>

^e  <https://orcid.org/0000-0002-3452-553X>

¹<https://www.construct.net/en>

²<https://www.stencyl.com/>

³<https://unity.com/>

⁴<https://www.unrealengine.com/en-US/>

2017). We defined a Domain-Specific Language (DSL) called UGameFeature that allows game designers to define their games in terms of game components without having to worry about implementation (programming) issues. We also defined an intermediate metamodel that represents the facilities of the C#-based scripting language supported by Unity. The structural differences between UGameFeature and the scripting language have been accommodated by a model-to-model transformation that we have also defined. Finally, we defined a model-to-text transformation that generates game scripts that can be executed by the Unity engine. This approach makes it possible to go from game design to game implementation through a chain of automated steps, with potential productivity benefits. This paper also discusses the practical obstacles of performing these steps and give recommendations to practitioners who want to apply MDE techniques in their projects.

This paper is further structured as follows: Section 2 describes the methodology applied in this work, by introducing the artefacts and techniques, Section 3 presents and justifies the metamodels we developed, Section 4 discusses our transformations, Section 5 discusses the transformation chain obtained by combining the metamodels and the transformations and how this transformation chain has been validated, Section 6 discusses some related work and Section 7 gives our conclusions and identifies topics for future work.

2 METHODOLOGY

This work has originated from a project that has been performed by a group of Master students of a Model-Driven Engineering course. This project has aimed at developing an initial solution to offer proper abstractions to game designers and at the same time automate code generation. Our goal has been to support the most relevant abstractions and to produce playable games as prototypes, not complete game products.

Our solution starts with a model of a game and generates code that can be executed by the Unity engine. To achieve this, we developed the following artefacts:

- The *UGameFeature Domain Specific Language* (DSL) that allows game designers to describe games. Since this is a simple proof-of-concept only the UGameFeature metamodel has been fully defined, which means that the concrete syntax of this DSL has been ignored.
- The *GameProgram metamodel* that serves as an

intermediary component between the game design and the game implementation.

- The *U2P model-to-model transformations* between the UGameFeature models (source) and the GameProgram models (target).
- The *M2T model-to-text transformation* that generates the Unity script code from GameProgram models.

This project has been implemented using the Eclipse modelling tools. Both metamodels have been specified using the Ecore tools in the Eclipse Modeling Framework, the model-to-model transformation has been written in the QVT Operational Mappings language and the model-to-text transformation has been written in Aceleo. Each artefact has been properly tested before being integrated in the final solution.

Figure 1 shows the transformation chain of the project. A UGameFeature model that is an instance of the UGameFeature metamodel is given as input to the U2P model-to-model transformation, which generates a GameProgram model that is an instance of GameProgram metamodel. The GameProgram model is then used as input to the model-to-text transformation, which generates code in the scripting language supported by Unity.

Each artefact of our solution is further discussed in the remaining sections. These artefacts can be found at <https://gitlab.utwente.nl/m7700446/ugamefeature-mde>.

3 METAMODELS

This section presents and justifies the metamodels defined in this work.

3.1 UgameFeature metamodel

Although some game design languages already exist, as far as we could observe they are too close to the programming level, so they are difficult to understand by game designers with limited programming skills. This inspired us to define the UGameFeature DSL, which we discuss here in terms of its metamodel.

UGameFeature metamodel has been defined to represent actions and components. Two main goals have guided the definition of the UGameFeature language elements: we should be able to define the most basic games, like e.g., 'Super Mario', using this language, and the language should be easy to use by game designers, i.e., the language should be intuitively appealing to them.

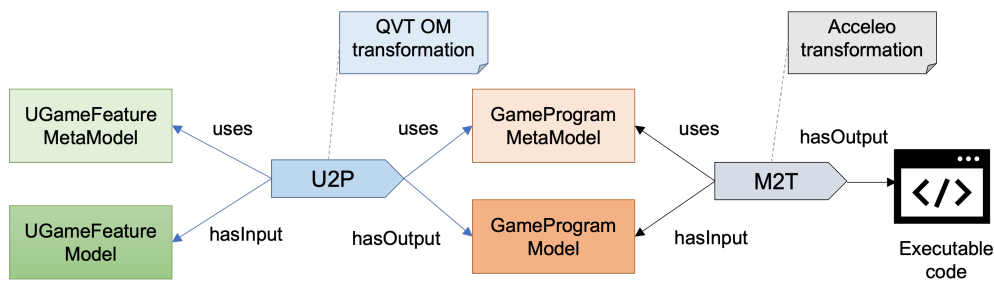


Figure 1: Transformation chain

The main metaclasses of this language are *GameObject*, *GUIelement* and *DataManager*. A *GameObject* in a game can be used to represent the player, the enemies, the stage and everything that is part of the game world. A *DataManager* contains *Data* elements, which are the global information concerning the game, like lives and scores, which are used in all types of games. A *GUIelement* is not necessary, but it is usually paired with a *Data* object to show information to the player.

We also represent the actions of the game, which are the global game events that are not necessarily triggered by the player. In our metamodel, we defined basic actions like the creation and destruction of objects, addition of force to an object, and changes to global variables. A *Movement* encapsulates actions that allow the objects to move. Two types of movements have been defined, namely platformer and top-down. A *Collision* is defined in a game object and can trigger an action. Figure 2 shows only the most important metaclasses of the *UGameFeature* metamodel, while the complete metamodel has 22 metaclasses.

Figure 2 shows that a *GameObject* can perform different types of actions, which can be distinguished by how they are related to the *GameObject* element. There are four alternatives, namely, through collision triggers, as a start or update relation, or via an *ActionKey*. *Start* and *Update* are methods of the Unity's *Monobehaviour* class, where start defined the actions to be executed when the object is created and update defines actions executed in every frame of the game. A collision trigger represents a contact between game elements, and it happens when an object touches another specific object. Finally, the *ActionKey* represents the traditional player interaction in which a player presses a button and something happens. In order to test the *UGameFeature* metamodel, we represented a simplified version of 'Super Mario' and a Minefield game with this metamodel by using the Sample Ecore Model editor. We concluded that the *UGameFeature* metamodel was suitable for repre-

senting the main aspects of these games.

UgameFeature is still preliminary. Its metamodel allows the definition of only small number of games with limited features. In addition, there is the design limitation that different *GameObjects* and different components cannot reference each other in any way. However, in our tests we designed complete games with objects that interact with each other and showed that it is possible to work around this limitation.

3.2 GameProgram metamodel

The *GameProgram* metamodel has been defined to capture the C# main programming concepts. Figure 3 shows the most important metaclasses of the *GameProgram* metamodel, while the complete metamodel has 20 metaclasses. Although it has been inspired by the scripting language supported by Unity, the *GameProgram* metamodel was defined to be as independent as possible of any specific platform, so that it could be used to model any program written in a C#-based scripting language like the language supported by Unity.

Program is the root metaclass of this metamodel. A program consists of many *Files*, which in turn contain *Namespaces*, which define programming scopes. Inside a namespace, *InternalTypes* can be defined, which can be either a *Class*, *Struct*, *Interface*, *Enum* or *Delegate*. Each *InternalType* can be assigned to functionality in a specific way. For example, a *Containment* may contain members, functions and constructors. In case a *Containment* is a *Class*, it can extend a superclass.

To denote types not defined in the program represented by the model, the program can also refer to external types, which are imported from some namespace and have an identifier. This allows classes and interfaces to be imported, extended, and implemented.

In order to test the *GameProgram* metamodel, we represented the Unity scripts for the simplified 'Super Mario' and Minefield games as instances of this

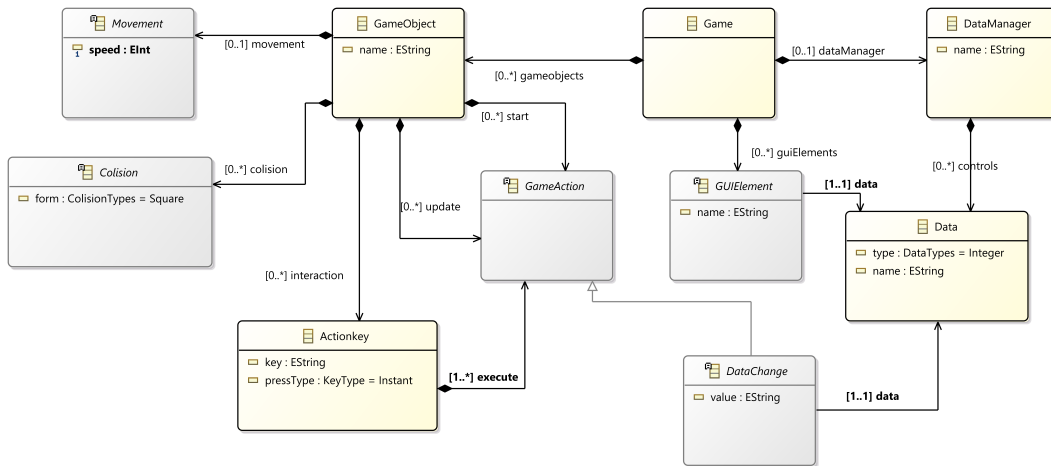


Figure 2: UgameFeature metamodel

metamodel, and translated these models initially by hand to the actual Unity scripts. We concluded that the GameProgram metamodel was suitable for representing the main programming aspects required to implement these games.

4 TRANSFORMATIONS

This section discusses the transformations developed in this project.

4.1 U2P Model-to-model transformation

The intermediate (GameProgram) metamodel was introduced to represent the programming concepts of a C#-based runtime environment, like the one supported by the Unity engine. This means that a model-to-model transformation had to be developed to solve the structural differences between the source (UgameFeature) and target (GameProgram) metamodels, while incorporating some implementation details.

Although the GameProgram metamodel is quite general in its representation of programming concepts, the U2P model-to-model transformation has been heavily centered around the support offered by Unity. This has been necessary since the resulting GameProgram model should match the capabilities supported by the Unity engine in order to be executable. The transformation was defined using the QVT Operational Mapping transformation language (Object Management Group, 2016). Figure 4 shows the outline of the transformation in the Eclipse

Overview. The transformation consists of 8 mappings, 3 helpers, 3 queries and 9 properties.

In this transformation, each UgameFeature model is transformed into one GameProgram model, which has a program as root element. Due to our use of Unity, we needed to create external types (as properties) for the native types of the Unity Engine namespace. Each game element (GameObject, GUIElement, or DataManager) of the source model is transformed into a File and a Class in the target model. Members and methods of a target class are created based on the game actions and game components of the element represented by the class. Each action or component can create many members, but the methods are usually already present from the MonoBehaviour structure extended by the game classes and so they only modify the body of their related methods. Methods are defined from the same relations that bring about the action types. Methods *start* and *Update* are both native methods for MonoBehaviour classes, and the collision methods are created by collision objects. From the action types, only ActionKey does not define its own methods, since it only extends the existing update method with an *if* statement. The processing of the components that only modify methods was done using helpers. For example, the code in Figure 5 represents the transformation of a UgameFeature Data into a get Method, to be included in the DataManager Class.

We debugged and tested our transformation with the models used to test the metamodels mentioned in Section 3, which means that we applied the U2P transformation to the UgameFeature model of the simplified Super Mario and minefield games, and compared the results with the GameProgram models of these games, respectively. In this way, after all er-

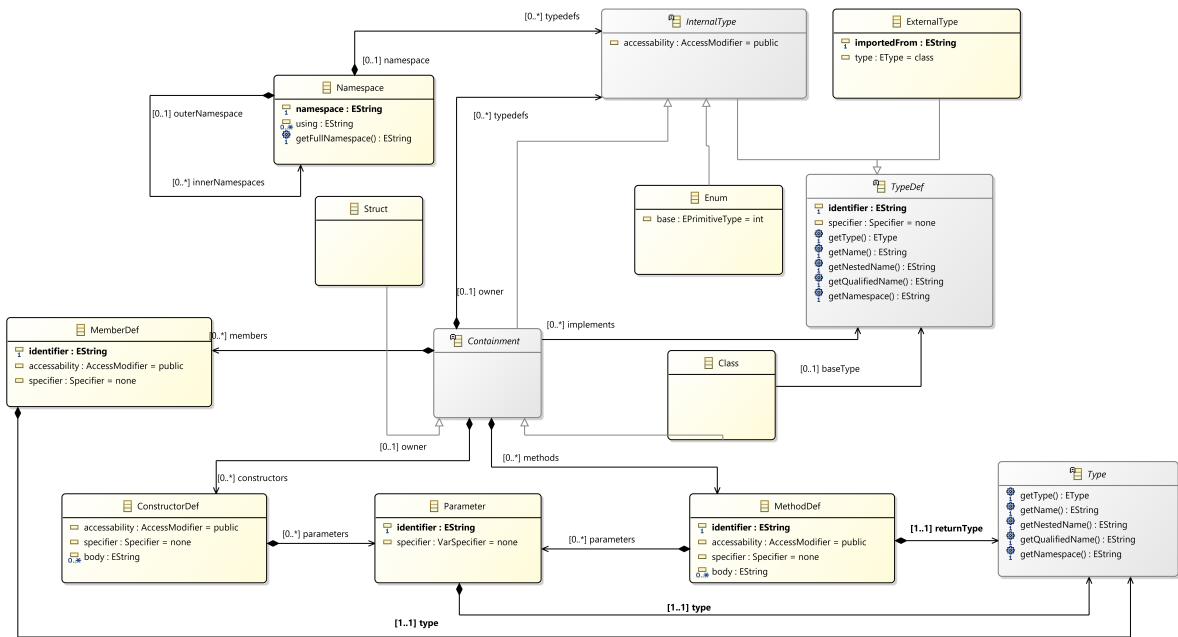


Figure 3: GameProgram metamodel

- UgameTransform
 - monoBehaviour : Prog::ExternalType
 - vector2 : Prog::ExternalType
 - rigidbody : Prog::ExternalType
 - text : Prog::ExternalType
 - collider : Prog::ExternalType
 - collision : Prog::ExternalType
 - gameObject : Prog::ExternalType
 - image : Prog::ExternalType
 - FIXED_NAMESPACE : String
 - main()
 - Unity::Actionkey::SetActionKeyBody(in upd)
 - Unity::Trigger::SetTriggerBody(in triggermet)
 - Unity::GameAction::Action2Body(in creation)
 - toType(in type:DataTypes)
 - toType(in _type:EPrimitiveType)
 - toType(in _type:TypeDef)
 - Unity::Game::Unity2Prog()
 - Unity::GameObject::GameObj2Class()
 - Unity::GUIElement::GUIElement2Class()
 - Unity::DataManager::DataManager2Class()
 - Unity::Data::Data2Member()
 - Unity::Data::Data2Get()
 - Unity::Data::Data2Set()
 - InternalType::Internal2File()

Figure 4: U2P model-to-model transformation outline

rors were removed, we confirmed that the target models generated by the U2P transformation indeed correspond to the intended ones, showing that the transformation has been correctly implemented.

```
mapping Unity::Data::Data2Get(): Prog::MethodDef{
  accessibility := AccessModifier::public;
  returnType := toType(self.type);
  identifier := "Get"+self.name.firstToUpper();
  body += "return this."+self.name + ' ';
}
```

Figure 5: M2M from Data to Method

4.2 Code generation

At this point, the GameProject model that represents a game program still needs to be serialised in terms of script code in order to be executed by the Unity engine. In order to obtain this script code, we defined a model-to-text transformation written in Acceleo⁵, which is an Eclipse-based code generation tool. Acceleo is therefore a proper match for the other artefacts developed in this project, particularly the GameProgram metamodel defined using Ecore.

Our Acceleo model-to-text transformation generates script code for the GameProgram model elements according to a template. Since the GameProgram metamodel is conceptually very close to the programming concepts of the runtime platform, this transformation ended up being quite straightforward. For example, the code in Figure 6 is the template that transform the Method class into code.

Once the model-to-text transformation was developed, the final step of the transformation chain could be tested. This has been done by generating code

⁵<https://www.eclipse.org/acceleo/>

```
[template public generateMethod(method: MethodDef)]
[generateFunction(method.accessability,
                 method.specifier,
                 method.returnType.getQualifiedName(),
                 method.identifier,
                 method.parameters,
                 method._body)/]
[/template]
```

Figure 6: M2T from Method to code

from the GameProgram models of the simplified Super Mario and Minefield games and importing the resulting code into Unity projects. This procedure has been used to debug the transformation, until no errors remained. Therefore these tests also served as a proof of concept for our transformation chain. The code generated with our model-to-text transformation uses some specific features of the Unity Engine, showing that the generated program is compatible with the Unity platform.

We are aware that these test procedures are not comprehensive, however we made sure children Game Objects and Prefabs could be used without problems. Since these are powerful capabilities, by enabling them in the code generation step we allow game developers to use some rather sophisticated Unity capabilities.

5 RESULTS

The final application was obtained by integrating the metamodels and transformations in a single transformation chain. Although we have not produced a fully encapsulated application that encompasses the whole transformation chain, we rely on the Eclipse facilities to perform the automated steps of the transformation chain.

To use our application, a game designer needs to define a model of a game using UgameFeature. Since we refrained from defining a concrete syntax, this model needs to be produced using the Ecore EMF editor. The definition of a concrete syntax is an obvious necessary improvement of the application, but it has been neglected for the time being since we were more interested in demonstrating the core of the transformation chain (the metamodels and transformations).

A game designer can then execute the U2P transformation on the game model in an Eclipse QVT OM project, producing in this way the GameProgram model. This can be done by running a QVT Operation Transformation Run Configuration in Eclipse that triggers the transformation with proper input and output models.

Once the GameProgram model is obtained, the game designer can run the Acceleo transformation

that is defined in an Eclipse Acceleo project by using an Acceleo Application Run Configuration that runs Acceleo with the template to generate script code using the GameProgram model as input. Alternatively, we could use Acceleo's facilities to trigger transformations using an Ant build or a Maven script. Since both transformations have been defined in the same platform, even if we have two separate projects for each transformation, respectively, the Acceleo project can refer to the output of the QVT OM project result. This means that it is not necessary to copy the resulting GameProgram model from one project to the other. It would be convenient to have this transformation streamlined within an one-click application, however we have not tried that as we focused on the core of the transformation chain.

Once the files with script code have been generated with Acceleo, the last step is to import them into a Unity project. The scripts have to be associated to the GameObjects while also adding the necessary components to match the restrictions described in the model, i.e., having a RigidBody or collision boxes. The game designer should keep notes of the names of objects, and more importantly, the tags of those objects as they are usually referenced in the code, since using names that do not match the names used in the model will lead to errors. This step creates a game in Unity as it was initially modelled by the designer.

Figure 7 shows a screenshot of a Unity project to which the code generated with our transformation chain has been imported. This project shows the simplified Super Mario game that we used to test the features of the project. The game designer defined the four game objects shown in the screen (wizard, bat, ghost, and amoeba) along with some game actions by instantiating the UGameFeature metamodel. Later, with the game scripts generated, the game designer can configure the specific type of the game objects, e.g., the wizard as Character and the others as Enemy. At the bottom of the Unity IDE are the scripts generated by our project (Bullet, Character, etc.) and at the left side, under SampleScene are the game objects that were modeled. This game is playable and works as intended, and has been generated without writing a single line of scripting code.

This project had some practical obstacles, the most notables were with the game metamodel and implementation choice. Concerning the game metamodel, it was challenging to define which of the game components were both necessary and sufficiently simple for creating games. Balancing those characteristics is an important factor when designing an abstract game metamodel. Concerning the implementation choices, we had the obstacle that many of the

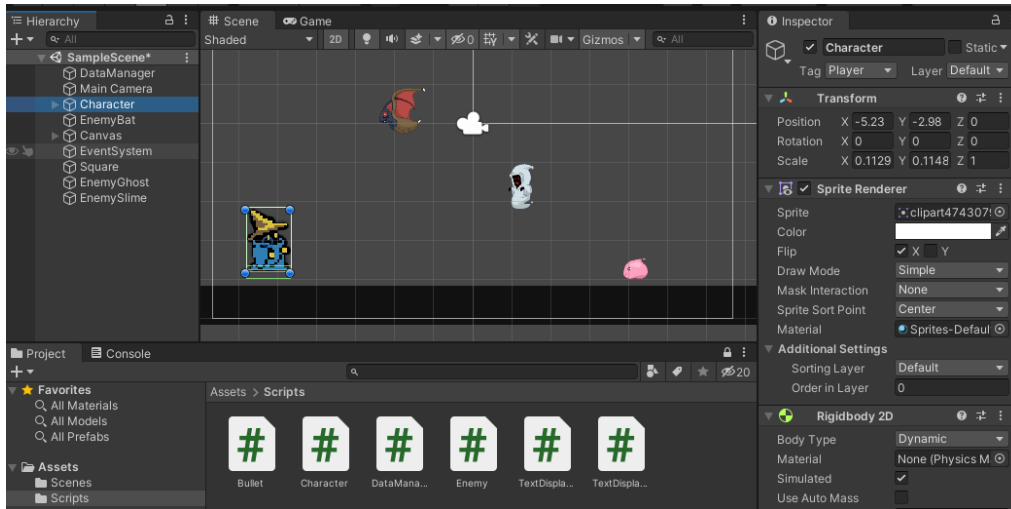


Figure 7: Unity project of the simple Super Mario game example

game features can be implemented in different ways, and determining the most suitable implementation for the automated creation was not simple. These challenges are the most stringent ones, not only for MDE application in game development, but for MDE-based projects in any application domain.

Although our approach presents an abstraction of the programming part of a game engine, the UGame-Feature metamodel to be used by game designers was inspired by the elements of the Unity game engine. One clear future direction is to define an even more abstract metamodel that can be conceived in collaboration with game developers, possibly through the development of a reference ontology of games. In addition, the development of metamodels of other game engines, along with the transformations to/from UGameFeature, can enable not only the code generation for the specific engines, but also facilitate the interoperability among them.

6 RELATED WORK

In this section, we discuss some related work that aimed at facilitating game programming by applying MDE. The recent literature review on Model-Driven Game Development (MDGD) (Zhu and Wang, 2019) identifies over 30 initiatives, and classifies them into some categories. The 'Use game engines or equivalent software' category is the most relevant for us, which covers four approaches based on Unity as main tooling environment. Although the approach for developing serious games with Unity (Aouadi et al., 2016) introduces an interesting DSL, it is oriented to pedagogical experts, representing concepts like ludic

resources and activities. Another approach for developing serious games that uses Unity (Matallaoui et al., 2015) extends the Gamification Modeling Language and provides transformations for JSON. Although it is a relevant work and generates 'ready-to-use' code (for an intermediary application), it only uses Unity as a study case and is oriented to gamification rather than gaming. The other two approaches are superficial and do not present implementation details.

GAMESPECT (Geisler, 2019) is a DSL with aspect-oriented programming features that supports game-specific DSLs, also providing a framework to assist game engine users to balance their games. This work overlaps with our approach in the attempt to automatically generate code to assist the developers of a specific engine (Unreal Engine 4), thus limiting the scope to this engine. Similarly, the Domain-Specific Game Development approach (Furtado, 2012) proposes a model-driven development methodology meant to allow game developers to efficiently develop games by applying software product lines, which enables the generation of games from the same 'family'. Differently from our approach, both approaches aim at assisting experienced developers, in contrast to novice ones.

The MDGD approach presented in (do Prado and Lucrecio, 2015) combines multiple DSLs with design patterns to provide flexibility and code generation integrated to manual coding for game developers. Although it has a purpose similar to ours, the major difference is that their application needs to be combined with manually created code, not removing the need to program, which has been our main goal. Another closely related work is the Casanova DSL (Abadi et al., 2015; Abadi, 2017; Di Giacomo, 2018), which

was created to reduce the cost for developing games, making coding easier for game developers. However, this approach still offers a programming language that requires its users (game designers) to write programs.

We observed that each of these initiatives reported a positive impact on the productivity of game development by using MDE. The main difference from our approach is that they have aimed at assisting developers to obtain safer and better code, whilst our main goal has been to abstract from programming issues so that game designers are able to create games without the need of writing *any* code.

7 CONCLUSIONS

MDE-based game development is not novel. The main difference of our approach to others is that we offer an abstraction of game programming aiming at assisting novice game designers in creating their games, without the need of changing the generated code. Although this work is limited to the scope of game development for the Unity engine, the results presented here offer ample opportunities for reusability, since we believe that they can be easily ported to other engines and/or development methods. Therefore, this approach has also potential for facilitating interoperability of game engines.

In this paper, we claim that MDE can enable game development without any manual code implementation, and that MDE allows faster creation of games. Further, we argue that the UgameFeature DSL is simple enough for novice users to understand, yet useful enough to represent simple games. Although in this project we showed that this DSL allows the representation of some popular (simple) games, future work should include a usability evaluation with professional game developers.

UgameFeature has limitations and its metamodel should be improved to address them. The most important is the metamodel simplicity, as it allows to model only a limited number of games. For example, right now it is not possible to model multiple levels, changing scenarios or animations. An improved version of this language with these capabilities will allow designers to model more elaborated games.

Alternatively, one could define an even more abstract metamodel with concepts that are not necessarily inspired by game engines, but in collaboration with game developers, possibly through the development of a game ontology. Instances of this ontology could then be transformed to models that can be used to generate code that runs in some game engine, as the UGameFeature metamodel that allows the game

to be implemented in Unity. This discussion leads to a more fundamental question in MDE, which is how to determine the number of intermediary models that are required to transform an abstract model that is sufficiently close to the 'world of the domain expert' (game designer in this work) to the most concrete code that can be executed in some runtime environment.

REFERENCES

- Abbadi, M. (2017). *Casanova 2, A domain specific language for general game development*. PhD thesis.
- Abbadi, M., Di Giacomo, F., Cortesi, A., Spronck, P., Costantini, G., and Maggiore, G. (2015). Casanova: A simple, high-performance language for game development. In *Joint International Conference on Serious Games*, pages 123–134. Springer.
- Aouadi, N., Pernelle, P., Ben Amar, C., Carron, T., and Talbot, S. (2016). Models and mechanisms for implementing playful scenarios. In *2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA)*, pages 1–8.
- Brambilla, M., Cabot, J., and Wimmer, M. (2017). *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2nd edition.
- Di Giacomo, F. (2018). *Metacasanova: a high-performance meta-compiler for domain-specific languages*. PhD thesis.
- do Prado, E. F. and Lucrecio, D. (2015). A flexible model-driven game development approach. In *2015 IX Brazilian Symposium on Components, Architectures and Reuse Software*, pages 130–139.
- Furtado, A. W. B. (2012). *Domain-Specific Game Development*. PhD thesis.
- Geisler, B. (2019). *GAMESPECT: A Composition Framework and Meta-Level Domain Specific Aspect Language for Unreal Engine 4*. PhD thesis, Nova Southeastern University.
- Matallaoui, A., Herzig, P., and Zarnekow, R. (2015). Model-driven serious game development integration of the gamification modeling language gaml with unity. In *2015 48th Hawaii International Conference on System Sciences*, pages 643–651.
- Object Management Group (2016). Meta Object Facility (MOF) 2.0 – Query/View/Transformation specification. Standard formal/2016-06-03, Object Management Group.
- Toftedahl, M. (2021). Which are the most commonly used game engines? Available on the site <https://www.gamedeveloper.com/>.
- Wijman, T. (2021). Global games market to generate \$175.8 billion in 2021. Available on the site <https://newzoo.com/>.
- Zhu, M. and Wang, A. I. (2019). Model-driven game development: A literature review. *ACM Comput. Surv.*, 52(6).