Engine- Cooperative Game Modeling (ECGM): Bridge Model-Driven Game Development and Game Engine Tool-chains

Meng Zhu
zhumeng@idi.ntnu.no

Alf Inge Wang
alfw@idi.ntnu.no

Hallvard Trætteberg
hal@idi.ntnu.no

Department of Computer and Information Science, Norwegian University of Science and Technology
Trondheim, Norway

ABSTRACT

Today game engines are popular in commercial game development, as they lower the threshold of game production by providing common technologies and convenient content-creation tools. Game engine based development is therefore the mainstream methodology in the game industry.

Model-Driven Game Development (MDGD) is an emerging game development methodology, which applies the Model-Driven Software Development (MDSD) method in the game development domain. This simplifies game development by reducing the gap between game design and implementation. MDGD has to take advantage of the existing game engines in order to be useful in commercial game development practice. However, none of the existing MDGD approaches in literature has convincingly demonstrated good integration of its tools with the game engine tool-chain. In this paper, we propose a hybrid approach named ECGM to address the integration challenges of two methodologies with a focus on the technical aspects. The approach makes a run-time engine the base of the domain framework, and uses the game engine tool-chain together with the MDGD tool-chain. ECGM minimizes the change to the existing workflow and technology, thus reducing the cost and risk of adopting MDGD in commercial game development. Our contribution is one important step towards MDGD industrialization.

Author Keywords

Model-Driven Development; Game Engine Game Development.

CONF '22, Jan 1 - Dec 31 2022, Authorberg.

unique doi string will go here

INTRODUCTION

Engine-based game development is the mainstream methodology today for commercial games, where the game engine as the central tool provides both the low-level technical implementation and the platform for development and management of high-level artifacts. Model-Driven Game Development (MDGD) on the other hand is an emerging research field, which brings the Model-Driven Development (MDD) methodology into the game development domain, following the model-centered development philosophy. It is interesting to compare the two methods from modeling perspectives: Game engines usually come with a tool-suite, with which game data such as world layout and characters can easily be created or modeled. If we look on game data as a *model*, its meta-model, which specifies the game domain, is implied in hand-written code. In MDGD, such a meta-model is explicitly created using a language workbench. Thus the game instance can be modeled through a domain-specific language specified by the meta-model. The game instance is either an executable model supported by a run-time engine, or a non-executable model that can be transformed into executable code running on top of a framework. In both cases, the software supporting the game model is in MDGD called a *domain framework* [1].

Engine-based game development has a long history in the game industry and it has impacted many aspects of commercial game development, such as technology, developer expertise and process. MDGD has to commit to current practice and embrace engine-based development to be practically useful. Integrating MDGD with game engine-based development is therefore an important step towards MDGD industrialization. However, if we look into the existing MDGD approaches in literature, gaps still remain. Some MDGD approaches overlook the whole game engine part and play an exclusive role in the game development [2, 3, 4]. Others use run-time game engines as the base of a Domain Framework while overlooking the game engine tools [5-9]. A common problem with these approaches is that they do not fully utilize the power of game engines, thus more or less reinvent the wheel by re-implementing

either a library or tools (or both) that are already provided by the game engines. This can lead to several risks preventing MDGD from industrialization:

1) *Tool maturity*: Game engines are complex software and are usually developed by professional vendors with a long development cycle. Replacing the functionality of game engines with MDGD tools will bring concerns to whether the MDGD tools can reach the maturity of game engines within a reasonable budget, while providing full MDGD support at the same time.
2) *Developer expertise and knowledge base:* Game engines have been used in the game industry for many years and it is easy to hire staff with relevant experience. Moreover, popular engines such as Unreal and Unity have a large and active community where it is easy to find existing solutions for various issues. It is expensive to train game developers and build a comparable knowledge base for MDGD tools.
3) *Workflow*: Game engine-based development has important impacts on the development workflow accepted by the industry. Moving the entire team out of the game engine-centered workflow is a large risk to take, seen from management point-of-view.

All in all, game engines and engine-based development are unique characteristics of the game industry that have to be considered when adapting general software engineering methodologies. This means that game engines must be organically integrated with MDD to establish a pragmatic MDGD approach. In this paper we propose Engine-Cooperative Game Modeling (ECGM), an MDGD approach that bridges the model-driven game development and game engines to minimize the modifications to the traditional methodology on one hand, while taking the advantage of modern MDD methodology on the other. With ECGM, the risk of applying MDD is reduced, thus MDGD having a better chance being accepted by game companies.

The rest of the paper is organized as follows: Section 2 introduces Model-Driven Development in general; Section 3 discusses some MDGD approaches in literature related to our work; Section 4 presents the ECGM approach; Section 5 briefly discusses a case study to show the usefulness of ECGM, and Section 6 concludes the paper.

MODEL DRIVEN DEVELOPMENT (MDD)

The central concept in the MDD domain is the *model*. In [10], a model is defined as "a simplification of a system built with an intended goal in mind". A similar definition can also be found in [11]. This means that theoretically many things are within the scope of a model. E.g. Favre argues in [12] that an article itself is a model: a model of the topic that it is about. However, the pragmatic scope of *model* is usually narrower. For example: Kelly and Tolvanen describe the developers understanding of models and code as "models are used for designing systems, understanding them better, specifying required functionality, and creating documentation. Code is then written to implement the designs" [1]. They distinguish between models and code, which is necessary for consolidating the theoretical base of MDD. Otherwise will MDD cover the traditional development methodology in terms of the provision that code is a kind of a model. Furthermore, in MDD, models are more than just documentation of the system: They can be transformed into a system, or they are the system themselves.

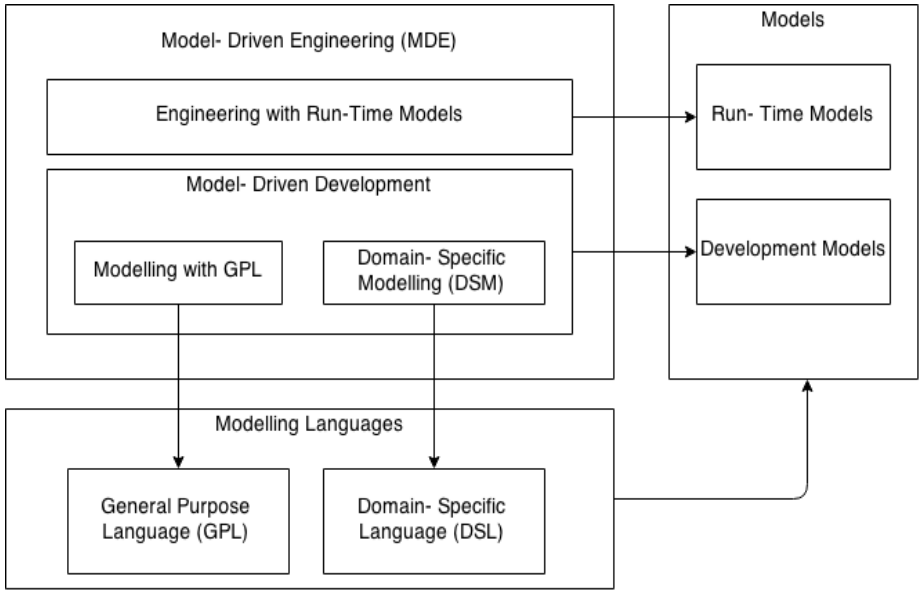
Model-Driven Development and Model-Driven Software Development (MDSD) are often used interchangeably in the software engineering community, although the former can express more than just *software* development. Various definitions of MDD have been proposed in the literature [13-16], where MDD can be defined as a software development methodology with following characteristics:

1) **Models are the focus**: MDD focuses on the models rather than the code, and the models are the major artifacts in the software development.
2) **Models are formal**: Models in MDD are formal thus can be transformed into software automatically or be executable.
3) **Models are on a high abstraction level**: Models are created with a modeling language at a higher abstraction level than a programming language, thus reducing the problem-solution gap.

A term similar to MDD is Model-Driven Engineering (MDE), which sometimes refers to the same concept as MDD in research papers, e.g. [17, 18]. But in [11], France and Rumpe enrich the meaning of MDE by distinguishing between the *development models* and the *runtime models*. These two models further lead to two directions in research with different research focus. The research focusing on development models is mainly concerned with *how can modeling techniques be used to tame the complexity of bridging the gap between the problem domain and the software implementation domain*, while the research focusing on runtime models is mainly concerned with *how can models cost-effectively be used to manage executing software*, and *how can models be used to effect changes to running systems in a controlled manner*? We agree with France and Rumpe who imply a broader scope than MDD, covering development of software by models as well as run-time management and control of software with models. Models in MDD can be created with either General Purpose Languages (GPLs), or Domain-Specific Languages (DSLs). UML is the most popular modeling GPL standardized by the Object Management Group [19], and was developed to be able to model all kinds of application domains [1]. However, people in the MDD community have pointed out some drawbacks of the language, e.g. [20]. Especially the one-size-fits-all design has been criticized for lowering the abstraction level [1]. The 2.0 version of UML supports semantic variation points and profiles as two forms of

extensions, which improved its domain appropriateness [21]. But the support for domain-specific abstractions is still weaker in UML compared to MDD approach Domain-Specific Modeling (DSM). DSM uses the modeling language developed for a specific application domain to solve the problems within a domain. DSM is claimed to have better domain appropriateness, restricted semantic scope, better support for generating code, and increased domain-specific reuse of components [22], and is also reported better than GPLs in regards to the improvements on software productivity [1]. DSM has also its drawbacks, where an important one is the potential huge investment from developing a DSL. Although the time to implement a DSL can be short, the expected time to benefit from it can decrease the investment interest [1]. An extensive consideration must be made on whether a new DSL for a problem should be created or not. In [23], the problem is discussed through identifying a set of decision patterns from the cases of applying DSM. Figure 1 shows a summary of the concepts of MDD, and their relationships.

Figure 1. MDD Concepts and Their Relationships



We have discussed the conceptual base of MDD. Another important aspect of MDD is its tool-chain that makes MDD practically useful. Model-Driven Software Development does not make sense without tool support [24]. Essential tools for MDD are tools enabling model execution, which includes two kinds of tools: the code generator and the model execution engine. Code generation is arguably the most widely used approach for doing MDD [24]. The interpreter is another mechanism supporting the model execution, which shares the same underlying principles as code generation [24]. The model editor is also a central tool in MDD [24], which is used for creating and maintaining models. When a model is created through a model editor, its correctness must be checked with respect to the meta-model before code generation. This checking is in most cases conducted with a model validator tool to avoid unnecessary complicated code generation [24]. Other tools, such as a model-level debugger, are also important but we will not get into these details in this paper.

MDD IN THE GAME DOMAIN

The game industry has actually a long history of using models. A typical example is the level model (or "world model") created with a level editor, which provides a visual environment for game world modeling in game engines like Unity and Unreal. However, the game elements that engine tools can model are restricted to a narrow scope of game software, which are mainly artistic and topological assets such as graphics and level structure. Other elements such as AI, control, and rules still have to be hand-coded, either with a General-purpose Programming Language (GPL) or with a scripting language. Some state-of-art game engines do provide visual modeling tools for creating script code, for example, Unreal Kismet [25], but these tools do not take full advantage of MDD, such as use of meta-models and language workbenches, which makes them difficult to adapt to changing requirements, or to new game domains.

To address the drawbacks of existing engine tools, researchers in the game community have proposed various MDGD approaches, and we will discuss some of them in this section. To make our discussion more focused, we only consider approaches where models are regarded as the central artifacts in game development, which must be described in a modeling language whose syntax and semantics are formalized.

One kind of MDGD approaches ignore the game engine while they tend to generate code directly based on the OS

or some kind of graphics SDK, for example [2, 3, 4, 26]. For simple games such as educational games or simple prototypes, this is a practical approach. However, it does not scale well for development of commercial games.

Other approaches use game engines or equivalent software components as a domain framework. Approaches in this category include [8] and [9] using Microsoft XNA[1], and [27] using the Corona SDK[2]. Moreover, some approaches modify game engines to promote them to a domain framework as suggested in [28], such as [5-7] that modified an open engine developed by DigiPen Institute of Technology (no longer available online), [28, 29] that modified the FlatRedBall engine[3], and [30] that added a layer (called "metaframework") to the GTGE engine[4]. These approaches use a run-time game engine, thus can provide important features such as 3D graphics and animation. However, the game engine modeling tools were not in scope of the approaches, thus they failed to take the full advantage of the game engines.

One approach uses a specific semantics engine to interpret the models at runtime [31, 32]. It can load and execute the model according to a set of semantic rules and render the game graphics as well as handling the player input. The semantics engine is a stand-alone software without any relationship to an existing engine. This means it is likely to lack some important features, or it has to make its own implementation.

Pleuß and Hußmann's approach [33-35] is the closest to our approach. They integrated MDD with authoring tools, more specifically Adobe Flash. Similar to our ECGM approach, two kinds of artifacts are generated: script code (ActionScript) and media objects (FLA files), and they are directly associated. The script code implements the game logic and the media objects can be edited with Adobe Flash tool. Their papers focus on their modeling language (MML) and technical details about the integration with the specific authoring tool. However, they do not make thorough discussion about the high level engineering approach, which is yet a major contribution of the paper in hand. Moreover, our paper discusses the integration with commercial game engines instead of general media tools, which further reduces the gap between MDGD and commercial game development.

THE ECGM FRAMEWORK

Figure 2 illustrates a typical engine-based game development architecture. The software in the double-edged boxes are usually third-party tools: Script Editor can be general text editors, or language-specific editors; DCC means Digital Content Creation, such tools include 3D Max, Photoshop, and so forth; GPL IDE is the general purpose programming language development environment, for example Visual Studio or Eclipse. The remaining components in the figure should be self-explanatory.

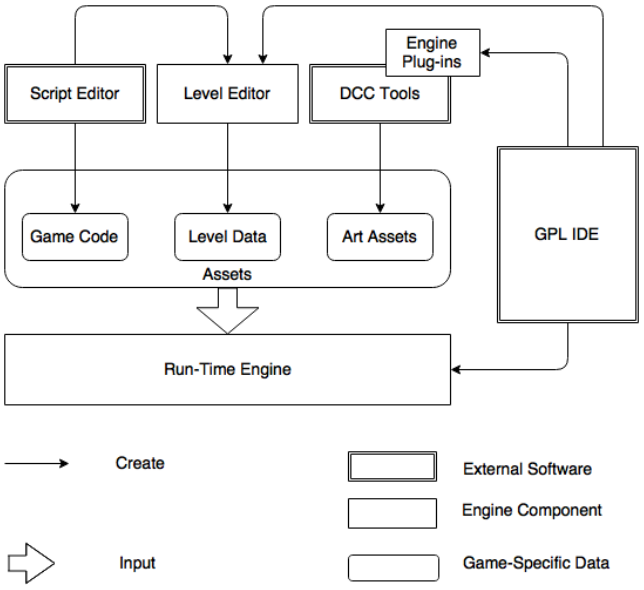


Figure 2. Architecture of Engine-Based Development

If we want to integrate the above-mentioned Engine-Based Development and a Model-Driven Development approach, we must find a place for MDGD in Figure 2. The existing MDGD approaches replace the script editor partly or entirely with a model editor and a code generator, with which the game models are created and (some of) the gameplay code is generated based on the models. However, such an approach overlooks the world data and the world editor. This means that how the model and the generated code relate to the world data is not resolved, leaving a gap between engine-based development and MDGD. Some approaches choose to re-implement the world editor to be a part of the MDGD tools. This is not a pragmatic approach as argued in Introduction section of this paper. If the MDGD tools and the engine tools are not aware of each other, the link between the model and the world data has to be built by the generated code and world level data, as the game developers do in traditional engine-based development. This raises at least two concerns:

1) The protocol for generating code, including the structure and names, depends on code generation software maintained by a programmer instead of a game designer. This means that a game designer cannot prototype or implement their ideas without help form programmers, as he or she does not know/understand all details of about the generated code to associate the model and the world data. A major advantage of MDGD is to allow game designers doing their work without help from programmers.

[1] Microsoft XNA: http://www.microsoft.com

[2] Corona SDK: http://coronalabs.com/products/corona-sdk/

[3] FlatRedBall Engine: http://newsblog.flatredball.com

[4] GTGE Engine: http://goldenstudios.or.id/products/GTGE/

2) When the code generation software is updated, the code structure and names can be changed. This will destroy the existing associations between the world data and the model, and rebuilding such associations can be very difficult in larger projects.

To solve the above problem, MDGD solutions must make the MDGD tools and engine tools interoperable to support a direct link between the model and the world data to achieve the following two goals: 1) the generated script code becomes transparent to the modeler and the game designer, eliminating the communication overhead, and 2) changes to the code generation software do not destroy the existing associations between the model and world data.

ECGM is a reference solution to this problem, and Figure 3 illustrates the architecture of ECGM. The topmost box in Figure 3 represents the MDGD meta-tools, which can be a language workbench or a collection of separate tools. The DSL developers use a meta-tool to create the meta-model, which constitutes the base of the game DSL. Two tools can be created with the meta-tool using a meta-model: the model editor, and the code generation/model transformation tool. With the model editor, gameplay developers, e.g. the game designers, can create the model defining the game-specific elements using the game DSL.

So far our approach is not different from the existing MDGD approaches. What distinguishes ECGM from other approaches is that *the game model will not only be transformed into script code, but also be transformed into world data*. The association between the game model and the world data is then generated within the process. When a game model is changed, corresponding changes will automatically be made in the world data by the code generation/model transformation tool, and game designers do not have to be aware of the generated code while solely focus on the game model and world data. This process is illustrated with an example below.

Assuming a scenario in a MDGD project where the level transition logic is modeled with a DSL, and the level layout is specified in a world editor. We illustrate the ECGM approach with the example shown in Figure 4, which is about the level transition of World 1-2 in the Super Mario Bros game.

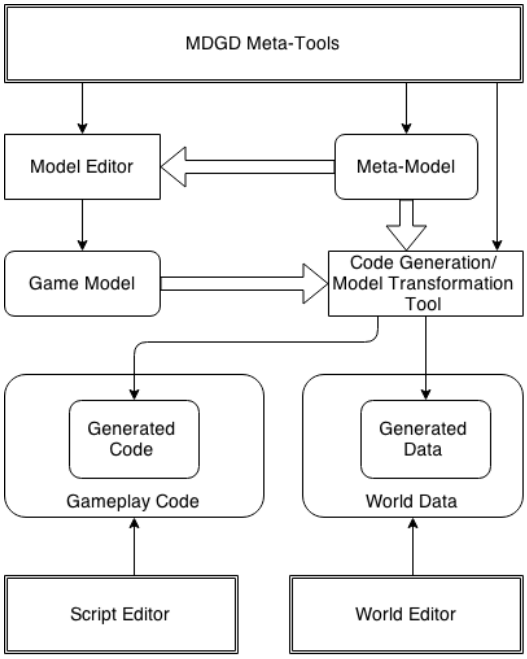


Figure 3. ECGM Architecture

Figure 4(c) shows a part of the scene of World 1-2 in a presumed world editor, and we label it "SWE". The scene shows three pipes, through which the player can exit World 1-2 and move on to World 2-1, 3-1, and 4-1 respectively. Figure 4(a) shows the DSL model of the level transition logic, where the double-edged circles embedded in the box labeled "World 1-2" represents three possible exits that connect to three world boxes. The model will be transformed into script code as shown in Figure 4(b), where a "class" (World_1_2) is generated for the scene (World 1-2). Three variables labeled *p1*, *p2* and *p3* are generated for the three exits in Figure 4(a) respectively. Since the DSL model does not contain any spatial information about the exits, it has to be specified in the SWE. *The DSL model editor and the SWE are isolated in this example.* This means that SWE holds no knowledge about the model, thus the associations between the location as well as the representation of pipes and their behaviors has to be implemented according to a custom protocol. For example, to build associations, manual operations in the SWE has to be done, which may be:

1) Create three "placeholder" objects in the SWE. One placeholder shown in Figure 4(c) is a red semi-transparent box overlapped with the pipe labeled "4".

2) Modify the properties of the placeholder object, set its name to "p1" so that the generated script can recognize the object and execute the expected code in certain conditions.

To do the task, the game designer has to know the protocol such as the name (p1) of the placeholder object, and the type of the placeholder object, which is implied in the generated code. This means that the game designer needs

knowledge about the script code. Moreover, once the code generation rule changes, the protocol may change correspondingly, e.g. it is possible that the name has to be "p_1" as the result of the change of code generation rule. The change then destroys the existing associations requiring extra job (change properties of all impacted placeholders in SWE). But if we choose to use ECGM, the process can be significantly simplified as illustrated in Figure 5.

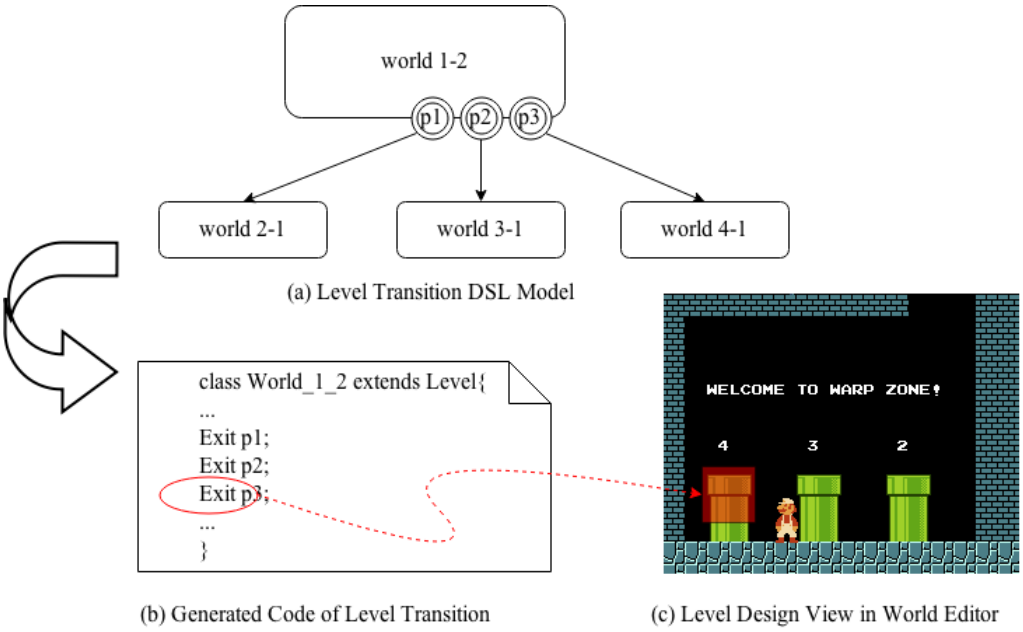


Figure 4. Level Transition DSL Model and Level Design View of World 1-2 in Super Mario Bros

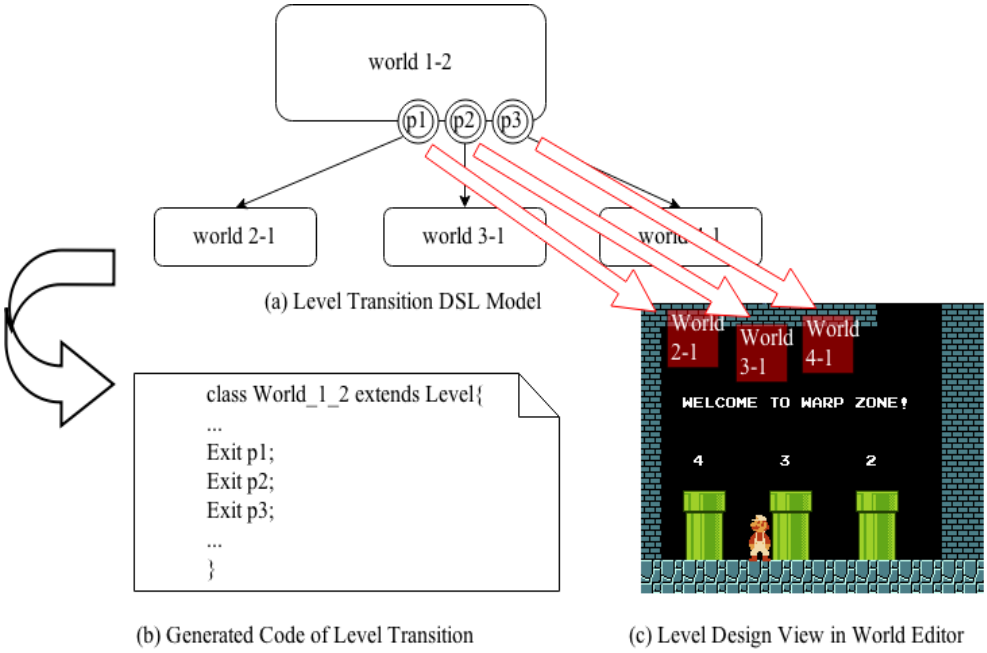


Figure 5. Level Transition Modeling with ECGM

In ECGM, the level transition model will not only be transformed into script code, but also be transformed into world data. Three placeholder objects will automatically be added to the world data after the model transformation, as shown in Figure 5(c). The remaining job for the game designer is to drag and drop the placeholder objects to the

right place and adjust its size if necessary. The properties of the placeholder objects have been set to reflect the relationship to the script code during code generation, so the generated script code is totally transparent to the game designer. Even the code generation rule can change in the future. The corresponding change to world data regarding placeholder objects will be done by re-doing the code generation. No manual modifications to the world data are needed, making the change irrelevant to the scene design.

The ECGM concept is easy to understand, but the implementation might be difficult. One major problem is that the world data is engine-specific, requiring different approach and effort for different engines. It can be very time-consuming to figure out how to manipulate the data format of the world editor. Having access to the source code of the world editor is important if the data is in a binary format. If we look ahead in the future of MDGD, it will be very helpful for the implementation of ECGM if game engine vendors provide open interfaces to their world editors. Next section further illustrates ECGM by presenting a case study: RAIL and Torque2D.

INTEGRATE RAIL WITH TORQUE 2D: A CASE STUDY

Reactive AI Language (RAIL) is a DSL we designed for modeling character behavior, i.e. the high-level AI of characters in action/adventure games. Due to paper length limitations, only a brief introduction to the main concepts will be given:

- **AI Pattern**: It represents the complete behavior of a specific character, including the character reactions to specific events in each particular state, and default behavior when no events occur. An AI Pattern may have 0 or multiple States.
- **State**: The "State" concept represents the current state of the NPC at a particular moment.
- **NPC Action**: The whole NPC behavior consists of many sequences of moves, and each completes a basic task. The NPC Action represents such a sequence of moves. Typical actions can be "move to a location", "shoot a bullet at the player", and "run to the player".
- **Event**: The action of NPCs is stimulated by an event or an event composite. An event can be directly connected to the player behavior, e.g. "the player enters vision", "the player becomes invisible", and "the player is aiming at me", or it can be related to other gameplay objects, e.g. "the boss is down" or "the light is off". An event can trigger an action, and/or other events, and the event-action chain depicts complex behaviors.

The abstract syntax and static semantics of RAIL are defined with an Ecore meta-model, and the language borrows some core concepts from State Machines. The top-level RAIL construct is *Game*, which is the container of all *AIPatterns* in a game. Each computer game to be modeled should have one and only one instance of *Game*. The *AIPattern* is the central construct of RAIL models that corresponds to the AI Pattern concept. Modeling with RAIL is mainly about creating various AIPattern instances, each of which defines a particular kind of NPC behavior. An AIPattern is stateful, meaning that the NPC reactions to events are influenced by the condition the NPC is in at a particular moment. The State is an abstraction of the condition in the domain description. Each AIPattern possesses a group of State instances reflecting all the possible conditions that are relevant to the reactions of the NPC following the AIPattern. But an AIPattern can only be in *one* State at a given moment, say the "Active State", and the initial Active State is named "default". A special case of an *AIPattern* is that it has only one state, then the state can be omitted and the Triggers (described later) will be directly connected to the *AIPattern*.

A State has a group of triggers, which defines what actions to perform in reaction to a stimulus that is typically an event or a composite of events. The Event construct can be further elaborated with *vision events*, *input events*, *AI interactive events*, etc. The stimulus can also be something other than Events, for example state change, pattern initialization, or logic operations. The Action construct encapsulates the actual actions to be performed by the AI pattern as the result of the stimulus. A common kind of actions is the *IssueCommand* which sends a specific command to the NPC controlled by the AI pattern, such as "Move to a Location", "Attack a Target", and "Look at a Place". The instances of a Trigger can be associated with a State, or directly associated with AIPatterns, where they become "Default Triggers". The Default Triggers will take effect in any State, and if they are in conflict with the State-owned triggers, they have the priority.

The concrete syntax of RAIL is based on a tree-view. We chose this form firstly because the AIPattern-State-Trigger-Action/Event hierarchical relationship naturally follows a tree structure, and secondly because with the Eclipse Modeling Framework you can get a tree-view model editor for free once a meta-model is defined with the Ecore language. Figure 6 shows a RAIL model within the Eclipse-based model editor.

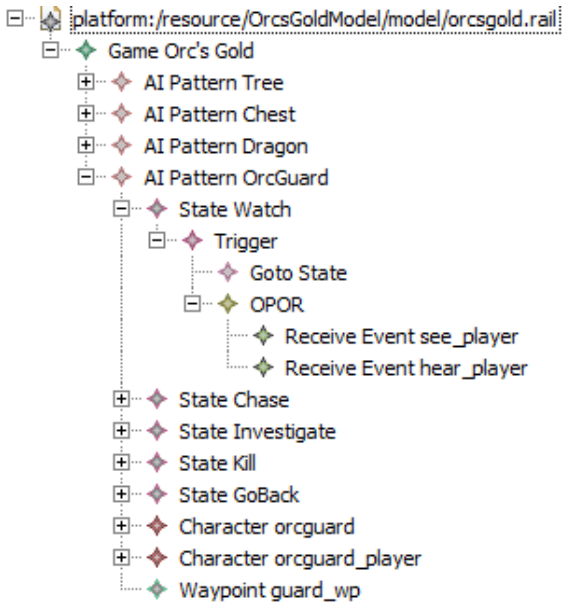


Figure 6. A RAIL Model in Editor

The implementation of the RAIL follows the ECGM approach, where Torque 2D is the target game engine to integrate. Torque 2D is a commercial game engine developed by GarageGames[1]. The game code for Torque 2D is written in "Torque Script", which has a C-style syntax plus some object-oriented features. Torque 2D engine provides a powerful world editor: the Torque Game Builder (TGB).

To integrate RAIL tools with the Torque 2D engine following the ECGM approach, Acceleo[2] was used to implement the code generation. Acceleo is an implementation of the Model to Text transformation Language (MTL) standardized by OMG, and it greatly reduces the effort of writing a code generator. Two kinds of artifacts were generated from the RAIL models: 1) The Torque Script code implementing the modeled behavior, and 2) the data for the TGB (world editor). The generation of Torque Script code is a trivial task: A Torque Script Class (it is similar to Class in C++ or Java) was generated for each AIPattern, and a couple of member functions were generated for the states and triggers possessed by the pattern. The Torque Script code must be associated with the graphical objects in the TGB. With the ECGM approach, the code-object relationships were built automatically through a specific generator, and the format of the generated data complies with the TGB extension protocol. The TGB uses an object palette to manage object prototypes. For each object prototype, e.g. a picture or a sprite animation, there is a visual object in the palette. Users can pick a visual object in the palette, and then create an object of the same type and initial attributes. The TGB extension protocol allows adding customized object prototypes to the palette of the world editor. We generated one pattern object prototype in the TGB palette for each AIPattern in the RAIL model. Thus, the AIPattern is visualized in the TGB as a graphical object like other built-in object prototypes. Figure 7 shows how the modeling tool and the world editor are working together.

If a user wants to connect *Pattern A* modeled with RAIL to *character A* in a level, he or she can drag Pattern A from the world editor palette to somewhere near the character in the level. The Pattern will automatically be linked to the nearest character, and the association is built by the generated code as well as the domain framework. In this case study, RAIL modeling tools are seamlessly integrated with the TGB, and the generated code is transparent to the game designers. We have prototyped the game *Orc's Gold* with ECGM. This game is a single player action game, where a player controls a human character who should steal a gold chest from orcs. We modeled four main patterns as it was presented in Figure 6. The modeling experience is quite convenient with the RAIL model editor as, with a few clicks and keyboard inputs, game designers can add a new model element. The editor also support drag and drop as well as copy and paste, which can make modeling even easier. We hand-coded a reusable domain framework on top of Torque 2D and the game-specific code is mostly generated from the RAIL model. The game-specific code should be the most cumbersome and error-prone to program, but with the ECGM approach this was avoided.

The initial investment (creating DSL and tool-chain) of model-driven development is a general concern. The solution of ECGM is to embrace engine tools narrowing the scope of modeling, thus reducing the effort needed to create the modeling language and tools, which was shown usefulness in our case study. The use of language workbench, i.e. EMF and Acceleo also significantly reduced the initial investment. Our case study showed that the initial investment on the meta-model and code generator for RAIL was acceptable, and the tools can be used to create many more patterns for making Orcs´ Gold into a real game, or be reused in other 2D action/adventure games.

When integrating RAIL with Torque 2D, we had to modify the engine tools. Although Torque 2D is relatively open, there are still some issues needed to be fixed at the source code level. The modifications we made were minor, but it introduces the problem that every time we update the engine, we have to redo the modifications. So for the engine vendors, it is wise to keep the format of level data flexible and open to external tools, because this can make the engines have better opportunity to be chosen as the target engine in MDGD projects.

[1] GarageGames: http://www.garagegames.com

[2]Acceleo official website: https://eclipse.org/acceleo/

Figure 7. Use AIPattern in the TGB

CONCLUSION

Today, the use of game engines is the mainstream approach for developing games. From a software architecture perspective, a modern game engine mainly consists of two parts: the run-time engine, and the world editor running on top of the run-time engine. To be practically useful, MDGD must be able to cooperate with both parts of a game engine.

Some existing MDGD approaches are aware of the game engine, however they have overlooked the cooperation with the world editor, leaving a gap between MDGD and the game engine. In this paper we have presented ECGM to address this gap. ECGM uses code generation or model transformation techniques not only to generate gameplay code, but also generate world data that can be manipulated in the world editor. With the ECGM approach, game designers working on world editors do not have to know the details of code generation, while they only have to manipulate the visual objects generated from the model.

The ECGM approach was demonstrated with a case study where Reactive AI Language, a DSL for action/adventure games was implemented and its toolchain were integrated with the commercial game engine Torque 2D. The integrated toolchain showed that the ECGM approach can make MDGD and engine based development feasible and convenient.

Further work may include integrating MDGD approaches with other commercial game engines to validate the feasibility and collect user feedback to evaluate the effectiveness of the approach. Developing more prototypes based on the integrated tool-chain can also provide usability data that is important for evaluating the approach.

REFERENCES

1. Kelly, S. and J.-P. Tolvanen, *Domain-Specific Modeling Enabling Full Code Generation*. 2008: John Wiley & Sons, Inc.
2. Reyno, E.M., et al., *Automatic prototyping in model-driven game development.* Comput. Entertain., 2009. **7**(2): p. 1-9.
3. Reyno, E.M. and J.A.C. Cubel, *Model-Driven Game Development: 2D Platform Game Prototyping*, in *Game-On 2008, 9th Int'l Conf. Intelligent Games and Simulation, EUROSIS*. 2008.
4. Altunbay, D., E. Cetinkaya, and M.G. Metin, *Model Driven Development of Board Games*, in *the First Turkish Symposium on Model-Driven Software Development (TMODELS)*. 2009.
5. Furtado, A.W.B. and A.L.M. Santos, *Using Domain-Specific Modeling towards Computer*

Games Development Industrialization, in *6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06)*. 2006.
6. Furtado, A.W.B. and A.L.M. Santos, *Extending Visual Studio .NET as a Software Factory for Computer Games Development in the .NET Platform*, in *2nd International Conference on Innovative Views of .NET Technologies (IVNET06)*. 2007.
7. Furtado, A.W.B., A.L.M. Santos, and G.L. Ramalho, *A Computer Games Software Factory and Edutainment Platform for Microsoft .NET*, in *SB Games 2007*. 2007.
8. Hernandez, F.E. and F.R. Ortega, *Eberos GML2D: a graphical domain-specific language for modeling 2D video games*, in *Proceedings of the 10th Workshop on Domain-Specific Modeling*. 2010, ACM: Reno, Nevada. p. 1-1.
9. Walter, R. and M. Masuch, *How to integrate domain-specific languages into the game development process*, in *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology*. 2011, ACM: Lisbon, Portugal. p. 1-8.
10. Bezivin, J. and O. Gerbe, *Towards a Precise Definition of the OMG/MDA Framework*. 2001.
11. France, R. and B. Rumpe, *Model-Driven Development of Complex Software: A Research Roadmap.*
12. Favre, J.-M., *Towards a Basic Theory to Model Model Driven Engineering.*
13. Hailpern, B. and P. Tarr, *Model-driven development: The good, the bad, and the ugly.* IBM Systems Journal, 2006. **45**(3): p. 451-461.
14. Mellor, S.J., A.N. Clark, and T. Futagami, *Model-Driven Development.* 2003.
15. Sendall, S. and W. Kozaczynski, *Model Transformation: The Heart and Soul of Model-Driven Software Development.* IEEE Software, 2003.
16. Selic, B., *The Pragmatics of Model-Driven Development.* IEEE Software, 2003.
17. Kent, S., *Model Driven Engineering*, in *IFM 2002*. 2002.
18. Schmidt, D.C., *Model-Driven Engineering.* IEEE Computer, 2006.
19. Group, T.O.M., *UML 2.0: Superstructure Specification. Version 2.0*. 2005.
20. Henderson-Sellers, B., *UML-the Good, the Bad or the Ugly? Perspectives from a panel of experts.* Softw Syst Model, 2005.
21. Krogstie, J., G. Sindre, and H. Jorgensen, *Process models representing knowledge for action: a revised quality framework.* Eur J Inf Syst. **15**(1).
22. France, R. and B. Rumpe, *Domain Specific Modeling.* Softw Syst Model, 2005.
23. Mernik, M., J. Heering, and A.M. Sloane, *When and how to develop domain-specific languages.* ACM Comput. Surv., 2005. **37**(4): p. 316-344.
24. Stahl, T., M. Voelter, and K. Czarnecki, *Model-Driven Software Development: Technology, Engineering, Management*. 2006: John Wiley & Sons.
25. *Visual Scripting Systems | Unreal Kismet*. [cited 2014 13/02]; Available from: http://www.unrealengine.com/features/kismet/.
26. Funk, M. and M. Rauterberg, *PULP scription: A DSL for Mobile HTML5 Game Applications.*
27. Marques, E., et al., *The RPG DSL: a case study of language engineering using MDD for generating RPG games for mobile phones*, in *Proceedings of the 2012 workshop on Domain-specific modeling*. 2012, ACM: Tucson, Arizona, USA. p. 13-18.
28. Furtado, A.W.B., et al., *Improving Digital Game Development with Software Product Lines.* Software, IEEE, 2011. **28**(5): p. 30-37.
29. Furtado, A.W.B., A.L.M. Santos, and G.L. Ramalho, *SharpLudus revisited: from ad hoc and monolithic digital game DSLs to effectively customized DSM approaches*, in *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*. 2011, ACM: Portland, Oregon, USA. p. 57-62.
30. Sarinho, V.T., et al. *A Generative Programming Approach for Game Development*. in *Games and Digital Entertainment (SBGAMES), 2009 VIII Brazilian Symposium on*. 2009.
31. Moreno-Ger, P., et al., *Language-Driven Development of Videogames: The <e-Game> Experience*, in *Entertainment Computing - ICEC 2006*, R. Harper, M. Rauterberg, and M. Combetto, Editors. 2006, Springer Berlin Heidelberg. p. 153-164.
32. Moreno-Ger, P., et al., *A documental approach to adventure game development.* Science of Computer Programming, 2007. **67**(1): p. 3-31.
33. Pleuß, A., et al., *Integrating authoring tools into model-driven development of interactive multimedia applications*, in *Proceedings of the 12th international conference on Human-computer interaction: interaction design and usability*. 2007, Springer-Verlag: Beijing, China. p. 1168-1177.
34. Pleuss, A. *MML: a language for modeling interactive multimedia applications*. in *Multimedia, Seventh IEEE International Symposium on*. 2005.
35. Pleuß, A., *Modeling the User Interface of Multimedia Applications, Model Driven Engineering Languages and Systems*, L. Briand and C. Williams, Editors. 2005, Springer Berlin / Heidelberg. p. 676-690.