RAIL: A Domain-Specific Language For Generating NPC Behaviors In Action/Adventure Game

Meng Zhu[1*][0000-0001-6639-8283] and Alf Inge Wang[1][0000-0002-5502-1138]
[1]Norwegian University of Science and Technology, Sem Sælandsvei 7-9, NO-7491, Trondheim, Norway
zhumeng@idi.ntnu.no, alfw@idi.ntnu.no

ABSTRACT. Domain-Specific Modeling (DSM) has shown its effectiveness of improving software productivity in many software domains [1], where Domain Specific Language (DSL) plays a key role. Also in the domain of video games, researchers have proposed various DSLs for developing different aspects of several game genres. This paper presents a DSL named RAIL for generating Non-Playable Character (NPC) behaviors in Action/Adventure Games. Our DSL borrows concepts from State Machines and adds some features to better suit the target domain. Further, we have implemented a tool-chain for RAIL using the Eclipse language workbench, and the tool-chain has been integrated with the level editor of the Torque2D game engine. To evaluate the DSL, we developed a prototype game and collected data regarding the development time and code lines. The results showed that RAIL significantly improves the productivity of developing NPC behaviors in the target game with a reasonable associated cost. In addition, the integration of the RAIL and the Torque 2D tool-chains provides a smooth development workflow.

Keywords: Game Development; Domain Specific Language; NPC Behavior.

1 Introduction

Domain-Specific Modeling (DSM) is an emerging software development methodology, which uses modeling languages specifically developed for a relatively narrow domain to model the problems within the domain. Further, either the solution is generated from the models, or the models are executable as (part of) the solution itself. In DSM, the Domain-Specific Language (DSL) plays a key role, which raises the language concepts to a higher abstraction level than General Purpose Languages (GPL) such as Java or UML, thus making the modeled solution simpler than using GPL.

Games are difficult to develop [2], and DSM can potentially reduce the complexity and cost of the development activities. DSM has shown its usefulness in developing software for many application domains [1], and we believe it also has special advantages for game domain, such as: higher abstraction level of models helps communication between technical and non-technical people in the cross-disciplinary team; DSLs use problem domain concepts thus allow game designers to implement gameplay without going through programmers; and DSM enables fast prototyping which is important in game development. Researchers have proposed some approaches adapting DSM to computer game domain, such as [3-5].

Note that "computer game" is a broad software domain, ranging from simple card games to massively multiplayer online games. It is impractical to create a DSL that supports *all* computer games simply because the number of language concepts will explode. Most of the existing DSM approaches have narrowed down the target domain to one game genre, e.g. Tower Defense [4] and 2D Platformer [6]. Some approaches further narrow down the scope to a game family or even a single game project, e.g. [7, 8]. Our DSL presented in this paper also targets specifically the Action/Adventure game genre. Moreover, it only focuses on the NPC behavior part of the entire game, which further narrows down the scope of the DSL.

Our DSL is named Reactive AI Language (RAIL), and it has borrowed the basic concepts from State Machines with some additional domain-specific features. We have implemented a tool chain for RAIL and integrated the tool chain with Torque 2D game engine. To evaluate the DSL and the tool chain, a prototype was developed and data on development effort was collected. The results showed that RAIL significantly improved the productivity in developing the prototype with an acceptable associated cost, and the integration of the RAIL and Torque 2D tool chains offers a smooth workflow.

The rest of the paper is organized as follows: Section 2 discusses related work; Section 3 presents the essential concepts of RAIL; Section 4 describes the design and implementation of RAIL and its tool chain; Section 5 presents the prototype to validate RAIL and discuss the results. Section 6 concludes the paper.

2 Related Work

Researchers have been exploring the potentials of DSM in game development in the recent years, and more and more model-driven approaches have been proposed in literature, such as [3, 16-19]. The major differences of the RAIL-based approach from the related work are on the target domain and the game engine-interoperability.

Some existing approaches ignore the game engine while they tend to generate code directly based on the OS or some kinds of graphics SDK, for example [20, 21]. Without the support from game engines, is it hard to support scalable game development. Other approaches use run-time game engines as domain frameworks, such as [22] and [23] use Microsoft XNA, and [24] uses the Corona SDK. Some approaches further modify game engines to promote them to a domain framework as suggested in [25], such as [26-28], [25, 29], and [30]. However, the game engine tools (world editor for example) have been ignored, thus they failed to take the full advantage of the game engines. The RAIL-based approach emphasizes the cooperation of game engine tools and MDD tools, making the non-technical game developers easier to work with, which is an important contribution of our work. Pleuß and Hußmann's approach [31-33] is the closest to our approach. They integrate MDD with authoring tools, more specifically Adobe Flash. In their approach, two kinds of artifacts are generated: script code (ActionScript) and media objects (FLA files). The script code implements the game logic and the media objects can be edited with Adobe Flash tool. Our paper discusses the integration with commercial game engines instead of general media tools, which further reduces the gap between MDD and commercial game development.

Regarding the target domain, many game genres have been explored by model-driven game development community, e.g. Platformer [6], RPG [34], Point & Click Adventure [5], and Pervasive Games [35]. Our approach focused on Action/Adventure which is not addressed in related work to our best knowledge. More importantly, we not only defined the genre of the target games, but also specified which part of the game is to be modeled. The target domain definition is therefore more systematic than most of the existing work.

State machines have been used as basis in several existing DSLs. E.g. it was used in [36] for modeling UI interaction, in [3] for modeling entity behaviors. [37] extends the general State Machine with adding domain-specific features such as hierarchical structure, parallel structure, and multi-interaction node for modeling narrative aspects of games. Our modification to the state machine is mainly adding trigger concept, which was proved effective in our prototyping.

3 RAIL: The Essential Concepts

RAIL is a behavioral DSL aiming at modeling the high-level AI of characters in action/adventure games. The behaviors to be modeled follow an event-reaction pattern, for example, the behavior of the Ghost (enemy actor) in Pac-Man (Namco, 1980). The Ghost behavior can be regarded as a state machine: The default state of the Ghosts is Patrol, where they randomly move around the map. If they receive an event that the player comes close, they will enter the Chase state, where they try to catch the player by running to him. While the player obtains a power-up anytime, he will have limited time to eat the Ghost. When the Ghosts receive the event about this, they will enter the Flee state and try their best to run away from the player.

We can identify some major concepts from the above description:

- AI Pattern: A specific kind of characters usually follows a behavioral pattern, determining what they are going to do when a given event is received in different conditions. We use the AI Pattern concept to denote such patterns, and each character in the target game is "controlled" by one AI Pattern.
- AI Pattern State: A specific event can trigger different behaviors for the same character when it is in different conditions. The "AI Pattern State" concept is the abstraction of the condition of the NPC at a moment. Typical examples of state include "chase" and "flee" in Pac-Man.
- Action: a meaningful character behavior consists of a sequence of moves, and each move completes a basic task. We use Action to denote the basic moves. Examples of Actions include "move to a location", "flee from the player", etc.
- Event: The action of characters is triggered by an event or an event composite, e.g. "the player enters vision", "the player becomes invisible", "the light is off". An event can trigger an action, and/or other events, and the event-action chain is the building block for complex behaviors.

RAIL is intended to support the modeling of the reactive AI in terms of the above concepts. Note that RAIL is built on top of the previous introduced concepts, while the internal details of the concepts is out of the language scope. E.g. given the Action "Walk to a place", it can be used as a building block in models, but how a NPC walk to a location involves a lot of low-level technologies such as path-finding, animation playback, which will not be modeled with RAIL. In another word, the problem domain of RAIL is restricted to the high-level AI of Action/Adventure games, and low-level technologies still must be implemented using traditional methods. Although the target domain sounds narrow, it is still valuable for practical development, because the high-level behavior is game-specific, and it is very difficult or even impossible to generalize it for various games. Implementation of the high-level behavior thus must be done in each individual project, which can take much of development resources. To address this problem, game engines provide scripting languages with some domain-specific support; for example, Unreal script supports the

"state" concept at the language level. However, RAIL as a dedicated DSL can further raise the abstraction level and make the solution even simpler.

4 DSL Design and Implementation

The abstract syntax and static semantics of RAIL are defined with a meta-model, which is specified using the Ecore meta-language provided by Eclipse Modeling Framework (EMF). We use the tree-view as the concrete syntax for RAIL, whose implementation, i.e. a RAIL model editor can be generated almost directly from EMF tools. Moreover, a code generator has been created using Acceleo, which is an eclipse based tool for code generation. Finally, the above RAIL tools are integrated with Torque2D game engine following the Engine Cooperative Game Modeling(ECGM) methodology [9]. ECGM is a model-driven game development methodology which emphasizes the interoperability of game engine tools and model-driven tools through model-transformations. We will detail the design and implementation of RAIL in the rest of this section.

4.1 RAIL Meta-Model

Figure 1 shows an excerpt of the RAIL meta-model, and some low-level details are omitted to fit it the page format.

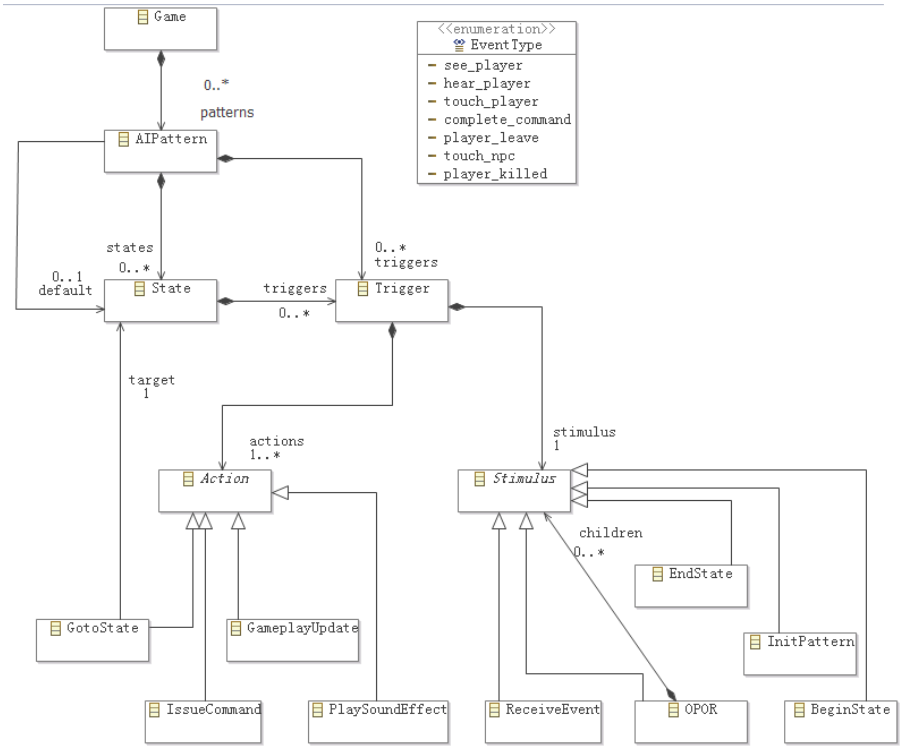

Fig. 1. RAIL Meta-Model

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. AIPattern is the central construct of RAIL models that corresponds to the AI Pattern concept as described earlier. Modeling with RAIL is mainly about creating various AIPattern instances, each of which defines a kind of NPC behavior. AIPattern is stateful, meaning that the reactions of NPCs to events are influenced by the state that the NPC is in at a moment. Here we borrowed concepts from state machines. The State construct denotes the state of AI Patterns. Each AIPattern possesses a group of States, but an AIPattern can only have one "Active State" at a given moment, and the initial Active State is the "default" state. A special case of 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 the actions to be performed in reaction to a stimulus that is typically an event or a composite of events. For example, a trigger can be:

Event("*See Player*") -> Action("*Move to The Player*")

Example 1

or be more complex as:

Event("*See Player*") OR Event("*Hear Player*") -> Action("*Alert Alliance*") AND Action("*Move to The Player*")

Example 2

The Event construct can be further elaborated with vision events, input events, AI interactive events, etc. The stimulus can also be something other than the Events, for example, state change, pattern initialization, and a group of basic stimuli connected with 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 will in effect send 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". Another kind of action is the Pattern Controller, which provides a way to manipulate the AIPattern at run-time, e.g. "Go to State" action sets the current State of the AI pattern to the one specified in the Action parameters.

The Triggers cannot only connect to States, but also directly associate with AIPatterns, and then they become "Default Triggers". The Default Triggers will take effect in any State, and if they conflict with the State-owned triggers, they have the priority.

RAIL is a DSL similar to a State Machine, which is a widely used design pattern [10] in gameplay programming. RAIL borrowed some concepts from the state machine, and made some important modifications to better fit it to the game modeling domain, such as:

- Defined domain-specific constructs to describe game domain concepts, such as *Actions* and *Events*.
- Provided *Trigger* concept, which can represent the reaction-based behaviors in a natural way.
- Supported global behavioral rules that will take effect in any state.
- Supported compact version of *AIPattern* with no state.

Note that RAIL is not intended to be a unified solution for modeling the gameplay of all the action/adventure games. Instead, it should be customized for each game families or even a single game project. But the high-level structure of the language, i.e. the constructs showed in Figure 1 should be reused without major modifications.

4.2 Tool Chain Implementation

The concrete syntax of RAIL is based on a tree-view. We choose this form because the AIPattern-State-Trigger-Action/Event hierarchy naturally follows a tree structure. Figure 2 shows a RAIL model example within the Eclipse-based model editor.

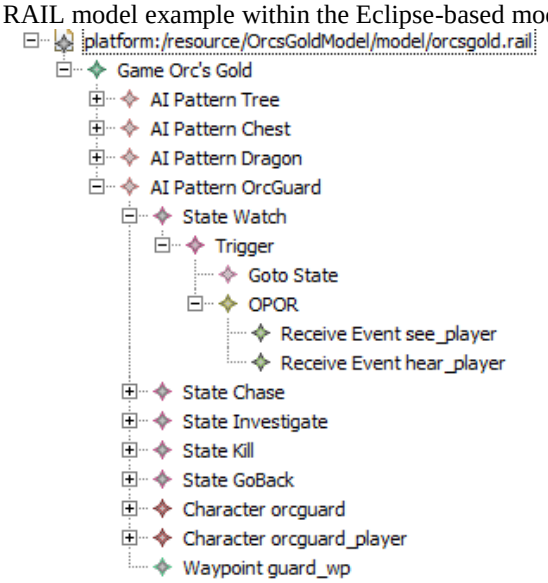

Fig. 2. A RAIL Model in Eclipse-Based Editor

With the Eclipse Modelling Framework (EMF), the model editor can be automatically generated from the meta-model. The code of the generated model editor is in java, and is deployed as an Eclipse plug-in, so that the tool is integrated with the Eclipse IDE. The java code can be modified for customization purposes, for example, changing the appearance of the language constructs, and optimizing the user interface. The default concrete syntax that the generated model editor provides is the Tree View, which is a perfect match for the structure of RAIL.

The code generator, on the other hand, requires much more work to create. There exist various frameworks on the Eclipse platform for code generation, such as Xtend and ATL. We chose Acceleo (https://eclipse.org/acceleo/). Acceleo provides a template language for creating code generators following the template and meta-model approach [11]. The code generator for RAIL was then implemented as a couple of templates, which took RAIL models as input and generated code in Torque script for Torque 2D game engine that we will discuss in next section. The code generation templates can be created and executed in the Eclipse platform since Acceleo is an Eclipse plug-in.

4.3 Integration of RAIL with Torque 2D Engine

Torque 2D is a commercial game engine developed by GarageGames (www.garagegames.com), which supports developing various genres of 2D games. Torque 2D provides a script language called "Torque Script" for developing game-specific code, which has a C- style syntax plus some object-based features. Moreover, Torque 2D engine comes with a powerful world editor: The Torque Game Builder (TGB). TGB organizes the game world through scenes (levels), and the scenes can be created in a WYSIWYG way.

There are mainly two steps in the process of integrating RAIL with Torque 2D: 1) raise the abstraction level of Torque 2D, and 2) implement the generator for script code and world data.

1. Raise the Abstraction Level of Torque 2D

Since RAIL was designed only for modeling high-level AI of Action/Adventure games, it targets a narrower domain and lies on a higher abstraction level than Torque 2D APIs. An abstraction layer must be implemented on top of the Torque 2D APIs to promote Torque 2D to a suitable domain framework [8].

The abstraction layer was implemented as a Torque Script library, where several concepts were implemented using an Object-Oriented approach. Character is the core module of the abstraction layer. Character simulates the creature or the machinery in a game that can perceive surroundings and react to the environment. Character is both an event source and an event handler: Character detects other objects at every frame, using the perception simulation algorithm, and generates corresponding perception events. The events are sent to the AI layer (modeled with RAIL) as the input for decision-making. On the other hand, a Character is also responsible for performing the commands sent from the AI layer, like move and attack. Other modules of the abstraction layer including input handling, event management, and global rules, which will not be discussed in detail in this paper.

2. Generate Code and World Editor Data for Torque 2D

To integrate RAIL tools with Torque 2D engine tools, two Acceleo projects were created: 1) Torque Script generator, and 2) TGB data generator.

Each RAIL model includes multiple AIPatterns, each of which defines a specific type of NPC such as neutral NPC, enemy soldier, and boss. The Torque Script generator generates a Torque Script class for each pattern, and a couple of member functions for the states and triggers possessed by the pattern.

The Torque Script code must be associated with the graphical objects in the TGB, and the code-object relationships were built automatically through a TGB data generator. The generator is also developed with Acceleo as templates, and the format of the generated data complies with the TGB extension protocol. The TGB uses an object palette to manage available scene objects. For each kind of scene object, e.g. a

picture, or a sprite animation, there is a visual object in the palette. The TGB extension protocol allows adding customized object prototypes to the palette of the world editor. We create one pattern object prototype in the TGB palette for each AIPattern in the RAIL model. Therefore, the AIPattern is visualized in the TGB as a graphical object like other built-in scene objects. When creating game scenes, designers access the AIPattern through the graphical objects in the TGB palette without knowing the existence of the generated code.

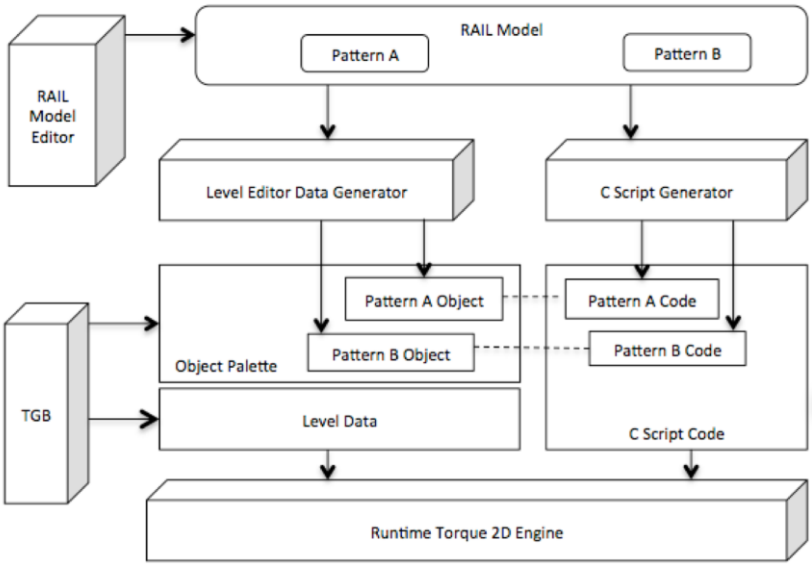

Fig. 3. The Tool Chain Architecture for RAIL Modeling with Torque 2D

4.4 Tool Chain Architecture

Figure 3 shows the tool chain architecture of RAIL. The dotted line between Pattern A in the object palette and Pattern A in C Script Code implies the association automatically built by the tool chain. 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 TGB palette to somewhere near the character in the level. The pattern will automatically link to the nearest character, and the association is built by the generated code as well as the domain framework. Figure 4 shows an example of using AIPattern in the TGB and the RAIL Editor.

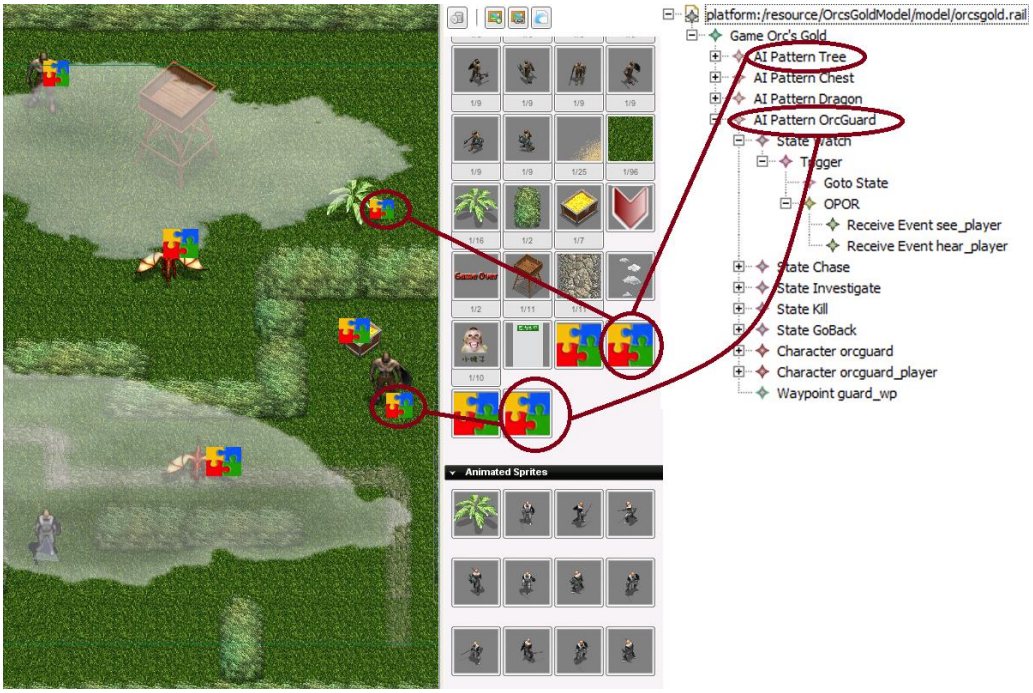

Fig. 4. Use AIPattern in TGB and RAIL Editor

5 Orc's Gold: A Prototype Game

To evaluate the RAIL as a game development tool, a prototype game was developed. The game is a 2D action/adventure game named *Orc´s Gold*. The game concept is that a player controls a human character who should steal gold chests from orc´s camps. The chests are guarded by orc guards and dragons, and they will try to kill the player if possible. The player can walk or run. When he is running, he moves faster than the orc guards so he can take the chests by wisely using the advantage of speed. If a player successfully steals all the gold chests on the map, he wins the game. Figure 5 shows a screenshot of *Orc´s Gold*.

Fig. 5. An In-Game Screenshot of *Orc´s Gold*

The game uses four AIPatterns to control the characters, which are Orc Guard, Dragon, Chest and Tree. The trees are purely decorative objects that almost have nothing to do with the gameplay, but they will play a "swing" animation when a player touches them. To evaluate the productivity impact of RAIL modeling, the four patterns were implemented by the first author using two methods: the manual coding method and the DSM method. The time used and Lines of Code (LOC) developed for the two methods are summarized in Table 1, from which we see that the domain framework including the low-level AI, input handling and other low-level mechanics took more time and lines of code to develop than the high-level AI Patterns. However, the domain framework is on a low abstraction level and less relevant to specific gameplay, it is reusable for future AI Patterns and even in future games.

Table 1. Comparison of Manual Coding and DSM in Developing *Orc's Gold*

	Time Used (Hours)		LOCs written	
Method	Domain Framework	AI Patterns	Domain Framework	AI Patterns
Manual Coding	20.5	**9.5**	1741	**357**
MDGD with RAIL		**0.7**		**0**

The time spent on the AI Pattern development is 9.5 hours with the traditional manual coding method, and it is dramatically reduced to 0.7 hours with the RAIL-based method. Regarding the LOCs, 357 lines of C-script code are used to implement the AI Patterns, and with MDGD method, all the AI Pattern code is automatically generated from a RAIL model, and no manual coding is needed. There is a significant productivity improvement from using RAIL, and the result is also in line with the reports of DSM from other software domains such as [12-14].

The benefits of DSM are not free, and the initial investment must be made for developing the DSL and the corresponding tools [15]. The time and LOCs used in creating RAIL and its tool chain are presented in Table 2.

Table 2. Cost of Developing RAIL and Its Tool-Chain

	Time (Hours)	LOCs
RAIL and Its Tool-Chain	4.5	278

As it was discussed in [15], the initial investment on DSM can be paid back by repetitively use of the DSL and the tools created, and by lowering the technical threshold for the developers. The more products and variants created with the DSL tools, the faster the investment is paid back. In the RAIL case, interestingly the investment is paid back in just one product: if we add the cost of developing RAIL tools to the cost of developing the prototype, the total cost is still lower than the manual coding method. This may because of two reasons: 1) the RAIL lies on a proper abstraction level that can significant improve the productivity while keeps the language simple for implementation and 2) the use of EMF and Acceleo framework significantly improved the productivity of developing the DSL tools

By analyzing the logic of the generated code and the manually written code, the performance of them is expected to be equivalent, because the algorithms and the mechanics implemented with two methods are identical. The results of a simple profiling also support the impression.

The RAIL-based modeling method also improves modifiability of the software. For instance, since the C-script does not fully support object-oriented programming, the manual code sometimes has duplicated parts spreading among several modules, and when the duplicated code needs to be changed, the same modifications must be done several times on different modules. This is an error-prone task. Thanks to the language features provided by Acceleo, the problem can be solved at the code generation level in RAIL-based modeling: the duplicated parts of the generated code can be generated from one code-generator module, and modifying the module will result in the modifications on all the generated modules with the duplicated parts. Generally, the language for writing code generator provides an extra means to compensate the drawbacks of the target language for modularizing the generated code well.

Modern game engines have provided various visual programming tools such as Unreal Kismet. Comparing to these tools, RAIL-based development requires less software engineering skills from the users, because the language concepts are closer to the game domain instead of the programming domain.

6 Conclusion

RAIL as a **domain-specific** language can specify behavioral aspects of Action/Adventure games. Prototyping Orc's Gold showed that RAIL and its tools can significantly reduce the time and code lines needed. Moreover, the cooperation of engine tools and MDGD tools offers an efficient workflow, which is benefit from the ECGM methodology [9].

The initial investment of model-driven development is of general concern. RAIL maximizes the interoperability of MDD and game engines, which can reduce the requirements to the MDD tools. The use of language workbench, i.e. EMF and Acceleo also significantly reduced the initial investment in the practical aspects. The case study showed that the initial investment on the meta-model and code generator for RAIL was acceptable, and it was paid back in just one project. Moreover, the tools can be used for creating many more patterns for extending Orcs´Gold to a real game, and even be able to be reused in other 2D action/adventure games.

Splitting the low-level AI and the high-level AI is necessary in game modeling. Script languages or GPLs are appropriate for implementing low-level AI, because they were just created for solving the problems on this abstraction level. The case study showed that low-level AI costs a lot of development effort, and this may raise a question that if RAIL has solved the difficult problems. But the low-level AI is reusable among patterns, even reusable among games, so the cost does not scale with the project complexity. The part of game that RAIL addressed can scale, which is more time-consuming in a real game project.

Further work includes extending RAIL to support larger scale games, and more game architectures, e.g. client-server, and applying it in more prototypes/games to get valuable feedback.

References

1. Kelly, S. and J.-P. Tolvanen, *Domain-specific modeling: enabling full code generation*. 2008: John Wiley & Sons.
2. Blow, J., *Game development: Harder than you think.* Queue, 2004. **1**(10): p. 28.
3. 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, Reno, Nevada*. 2010.

4. Sanchez, K., K. Garces, and R. Casallas. *A DSL for rapid prototyping of cross-platform tower defense games*. in *10th Colombian Computing Conference, 10CCC 2015, September 21, 2015 - September 25, 2015*. 2015. Bogota, Colombia: Institute of Electrical and Electronics Engineers Inc.
5. 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.
6. Reyno, E.M. and J. Carsi Cubel, *Automatic prototyping in model-driven game development.* Computers in Entertainment, 2009. **7**(2).
7. Van Hoecke, S., et al. *Enabling control of 3D visuals, scenarios and non-linear gameplay in serious game development through model-driven authoring*. in *5th International Conference on Serious Games, Interaction, and Simulation, SGAMES 2015, September 16, 2015 - September 18, 2015*. 2016. Novedrate, Italy: Springer Verlag.
8. Maier, S. and D. Volk. *Facilitating language-oriented game development by the help of language workbenches*. in *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*. 2008. ACM.
9. Zhu, M., A.I. Wang, and H. Trætteberg, *Engine- Cooperative Game Modeling (ECGM): Bridge Model-Driven Game Development and Game Engine Tool-chains*, in *ACE2016 the 13th International Conference on Advances in Computer* 2016: Osaka, Japan.
10. Gamma, E., et al., *Design patterns: elements of reusable object-oriented software*. 1994: Pearson Education.
11. Stahl, T., M. Voelter, and K. Czarnecki, *Model-Driven Software Development: Technology, Engineering, Management*. 2006: John Wiley \\& Sons.
12. Kieburtz, R.B., et al. *A software engineering experiment in software component generation*. in *Proceedings of the 18th international conference on Software engineering*. 1996. IEEE Computer Society.
13. Weiss, D.M., *Software product-line engineering: a family-based software development process.* 1999.
14. Kelly, S. and J.-P. Tolvanen. *Visual domain-specific modeling: Benefits and experiences of using metaCASE tools*. in *International Workshop on Model Engineering, at ECOOP*. 2000.
15. Kelly, S. and J.-P. Tolvanen, *Domain-Specific Modeling Enabling Full Code Generation*. 2008: John Wiley & Sons, Inc.
16. Furtado, A.W. and A.L. Santos. *Using domain-specific modeling towards computer games development industrialization*. in *The 6th OOPSLA Workshop on Domain-Specific Modeling (DSM06)*. 2006. Citeseer.
17. Guana, V. and E. Stroulia. *Phydsl: A code-generation environment for 2d physics-based games*. in *2014 IEEE Games, Entertainment, and Media Conference (IEEE GEM)*. 2014.
18. Matallaoui, A., P. Herzig, and R. Zarnekow. *Model-driven serious game development integration of the gamification modeling language GaML with unity*. in *48th Annual Hawaii International Conference on System Sciences, HICSS 2015, January 5, 2015 - January 8, 2015*. 2015. Kauai, HI, United states: IEEE Computer Society.
19. Reyno, E.M. and J.A.C. Cubel. *Model-driven game Development: 2D platform game prototyping*. in *9th International Conference on Intelligent Games and*

Simulation, GAME-ON 2008, November 17, 2008 - November 19, 2008. 2008. Valencia, Spain: EUROSIS.

20. Reyno, E.M., et al., *Automatic prototyping in model-driven game development.* Comput. Entertain., 2009. **7**(2): p. 1-9.
21. 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.
22. 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.
23. 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.
24. 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.
25. Furtado, A.W.B., et al., *Improving Digital Game Development with Software Product Lines.* Software, IEEE, 2011. **28**(5): p. 30-37.
26. 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.
27. 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.
28. 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.
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. Pleu\, 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.
32. Pleuss, A. *MML: a language for modeling interactive multimedia applications*. in *Multimedia, Seventh IEEE International Symposium on*. 2005.
33. 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.
34. Cutumisu, M., et al., *Generating ambient behaviors in computer role-playing games.* IEEE Intelligent Systems, 2006. **21**(5): p. 19-27.

35. Guo, H., et al., *Realcoins: A case study of enhanced model driven development for pervasive games.* International Journal of Multimedia and Ubiquitous Engineering, 2015. **10**(5): p. 395-411.
36. Pleuss, A. *Modeling the user interface of multimedia applications*. in *8th International Conference on Model Driven Engineering Languages and Systems, MoDELS 2005, October 2, 2005 - October 7, 2005*. 2005. Montego Bay, Jamaica: Springer Verlag.
37. Marchiori, E.J., et al., *A visual language for the creation of narrative educational games.* Journal of Visual Languages & Computing, 2011. **22**(6): p. 443-452.