Automatic Prototyping In Model-Driven Game Development

EMANUEL MONTERO REYNO and JOSÉ Á. CARSí CUBEL
Software Engineering and Information Systems Research Group,
Technical University of Valencia

Model-driven game development (MDGD) is an emerging paradigm where models become first-order elements in game development, maintenance, and evolution. In this article, we present a first approach to 2D platform game prototyping automatization through the use of model-driven engineering (MDE). Platform-independent models (PIM) define the structure and the behavior of the games and a platform-specific model (PSM) describes the game control mapping. Automatic MOFscript transformations from these models generate the software prototype code in C++. As an example, Bubble Bobble has been prototyped in a few hours following the MDGD approach. The resulting code generation represents 93% of the game prototype.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*Languages; Tools*; I.6.5 [**Simulation and Modeling**]: Model Development—*Modeling methodologies*

General Terms: Design, Documentation

Additional Key Words and Phrases: Model-driven engineering, model-driven game development, game development, game prototyping

ACM Reference Format:

Reyno, E. M. and Carsí Cubel, J. Á. 2009. Automatic prototyping in model-driven game development. ACM Comput. Entertain. 7, 2, Article 29 (June 2009), 9 pages.
DOI = 10.1145/1541895.1541909 http://doi.acm.org/10.1145/1541895.1541909

1. INTRODUCTION

The increasing complexity of game development [Blow 2004] highlights the need for tools to improve productivity in terms of time, cost, and quality. Since game prototyping and play-testing have been recognized [Henderson 2006; Waugh 2006] as fundamental engines of iterative game design, the automatization of software prototyping through model-driven engineering (MDE) is

Authors' address: Software Engineering and Information Systems Research Group, Department of Information Systems and Computation, Technical University of Valencia; email: {emontero, pcarsi}@dsic.upv.es.

DOI 10.1145/1541895.1541909 http://doi.acm.org/10.1145/1541895.1541909

proposed. This approach accelerates the design process and allows software prototype evolution towards the final game.

Model-driven software engineering (MDE) introduces a change of paradigm in which models become first-order citizens in the software lifecycle. These models become the base of software development, maintenance, and evolution. This article applies that change of paradigm to the lifecycle of games. In model-driven game development (MDGD), models become first-order elements in game development, maintenance, and evolution. By focusing on models that specify games instead of the code that implements games, a higher level of abstraction and automatization of the development process can be achieved.

MDGD has some commercial advantages to consider. First, the productivity and reusability of software artifacts increases, thereby reducing development time. Second, middleware that is specific to game development such as game engines continue to be encouraged. In addition, the interoperability and the portability between game platforms increase.

Using models to specify games also has academic advantages to consider. In the first place, it creates a common language between game developers and academics, which is a key element for game design study, communication, and evolution [Salen and Zimmerman 2004]. Second, it creates a set of standard tools for game design providing formal and abstract requirements [Church1999]. It also enhances iterative game design, following successive cycles of prototyping, play-testing, evaluation, and refinement [Salen and Zimmerman 2004; Fullerton et al. 2004]. As a first application of MDGD fundamentals, a tool has been developed to semi-automatically prototype 2D platform games from models.

The structure of this article is the following: Section 2 shows the state of the art in MDGD; Section 3 explores the main technologies that support the implementation of a tool for 2D platform game prototyping; finally, Section 4 details the conclusions and future lines of research.

2. STATE OF THE ART

Game prototyping has traditionally been used ad hoc [Waugh 2006] in order to reach the fundamental gameplay mechanics upon which to build the final game as soon as possible. Paper and software prototypes are discarded when validated, and then the game implementation starts from scratch. To date, no approximations to software prototyping automatization have been found. In MDGD, models are used as first-order elements to specify games (and prototypes), which allows automatic transformations to code.

With regard to game development automatization, the use of domain-specific modeling (DSM) was proposed by Furtado and Santos [2006]. This proposal uses graphical domain-specific languages (DSL) to represent the different aspects of games. Automatic code generation from the models is applied via the software factories approach. Similarly, MDGD proposes a change of paradigm in game development by raising the level of abstraction and using models as first-order citizens.

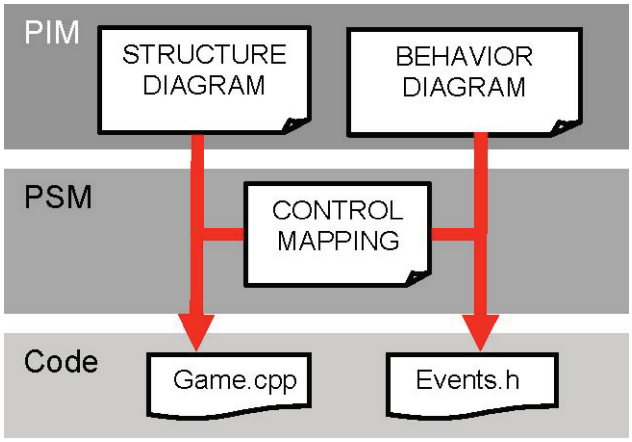

Fig. 1. Development scheme.

Model-driven development (MDD) has been proposed for interactive multimedia applications, with excellent results [Pleuß and Hußmann 2007]. Since videogames are interactive multimedia applications, these results can be considered precedents of MDGD. Additionally, by using a component-based approach [Folmer 2007], support for middleware that is specific to game development can be provided (such as the Microsoft XNA framework for multiplatform games).

3. DEVELOPMENT

As a practical MDGD demonstration, a tool has been developed using the Eclipse modeling framework, which semi-automatically generates 2D platform game prototypes from models.

Platform-independent models (PIM) define the structure and the behavior of the game without addressing the characteristics of the underlying technology platform (programming language, hardware architecture, etc.).

A platform-specific model (PSM) describes the control mapping, which associates the game actions to the controller signals. Finally, two MOF script transformations from these models generate the C++ code of the prototypes using the Haaf Game Engine. The 2D platform game prototype can be executed by editing and compiling the resulting code.

Each of the models and transformations (Figure 1) that implement the tool for 2D platform game prototyping is described in the following sections.

A classic 2D platform game (*Bubble Bobble*) is prototyped as an example. In this game, the player controls a dragon called Bub that blows bubbles at its enemies (Benzo) in order to trap them inside the bubbles. Bub can then jump on the bubbles to pop them, which destroys the enemies trapped inside. By finishing off his enemies this way, Bub gets rewards for extra points, and progresses to the next level.

In order to construct a prototype of this game, the visual metaphor of each game element can be simplified. The player character, the bubbles, the enemies,

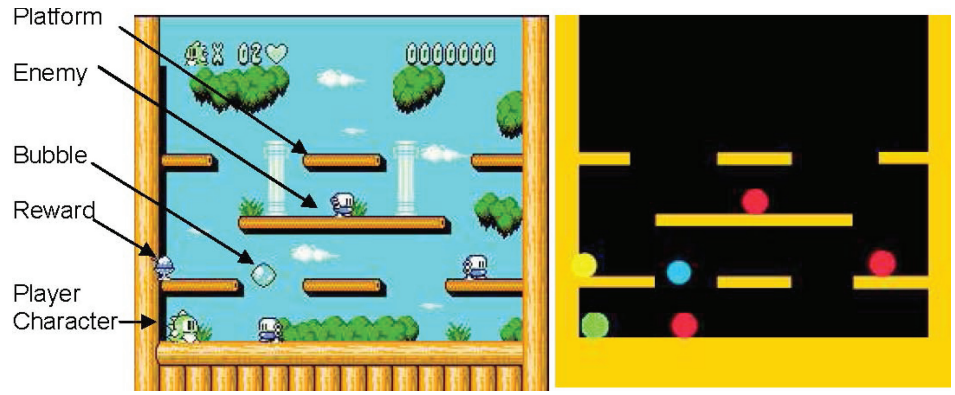

Fig. 2. Example 2D platform game (a) and its prototype (b).

the rewards, and the platforms (Figure 2(a)) can be represented as colored circles and squares (Figure 2(b)).

3.1 Structure Diagram

In this work, a class diagram extended with stereotypes is used to describe the structure of 2D platform games. Stereotypes are a way of expanding the Unified Modeling Language (UML) to create new model elements suited to the problem domain which, in this case, are 2D platform games.

The player character stereotype describes a game entity that is controlled by the player. The enemy stereotype denotes an internally controlled game entity that opposes the player character. The object stereotype is used to describe the other passive game entities. The concepts associated to each stereotype were defined in the compilation semantics, and introduced in the code generated by the automatic transformations.

In the *Bubble Bobble* example prototype, the structure diagram (Figure 3) details the following: a game level has a descriptive name associated to it; Its game entities are affected by gravity (attraction towards the ground) and friction (resistance to movement). The main game entities are the player character, Bub, the bubbles, the Benzo enemies, the rewards, and the platforms. The player character Bub has a number of lives and a score which can be increased by destroying Benzo enemies or collecting rewards. A bubble may or may not have a Benzo enemy inside it.

3.2 Behavior Diagram

State diagrams are used to describe the basic behavior of each game entity.

In the *Bubble Bobble* example prototype, the behavior diagrams show that the player character Bub (Figure 4(a)) can be alive in two states: in the air or on the ground or platforms. Bub can blow bubbles in both states, but can only walk and jump from the ground or platforms. If Bub collides with a bubble, it will pop. If Bub collides with a reward, he will get its points. If Bub collides with

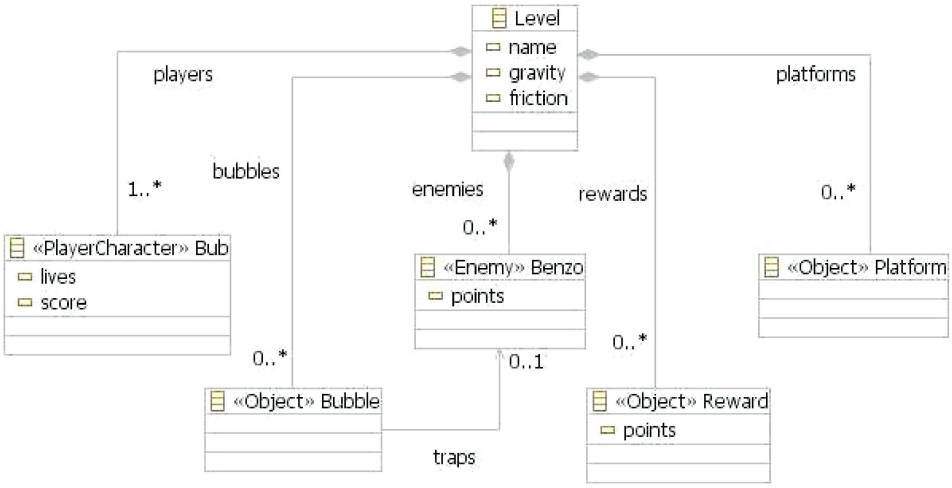

Fig. 3. Structure diagram of the *Bubble Bobble* prototype.

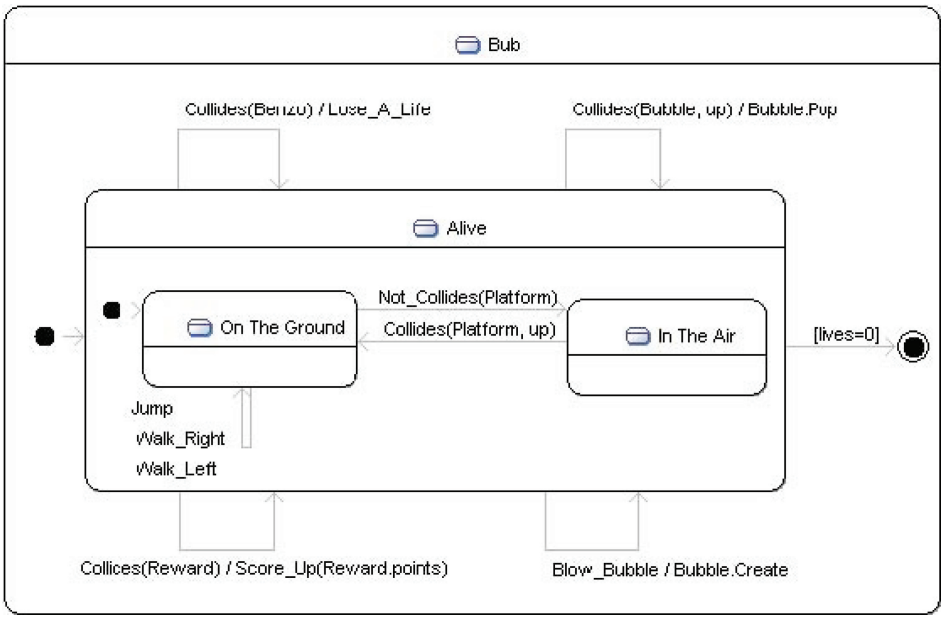

Fig. 4(a). Behavior diagram of Bub.

a Benzo enemy, he will lose a life. When Bub loses all his lives, he dies and the game is over.

The bubbles (Figure 4(b)) can be in three different states: blowing from the dragon's mouth, floating with a Benzo enemy inside, or floating empty. If a bubble collides with a Benzo enemy when it is launched, it will trap it inside. Then if the dragon Bub collides with a bubble, the bubble pops and the Benzo

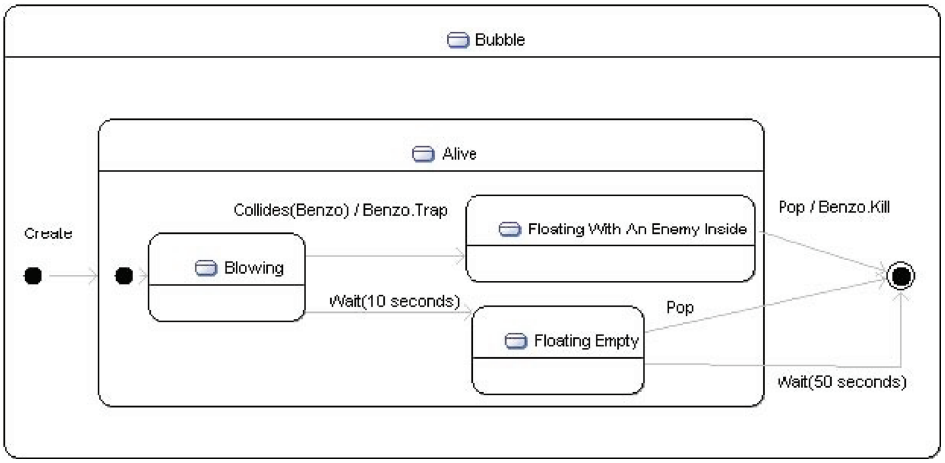

Fig. 4(b). Behavior diagram of a bubble.

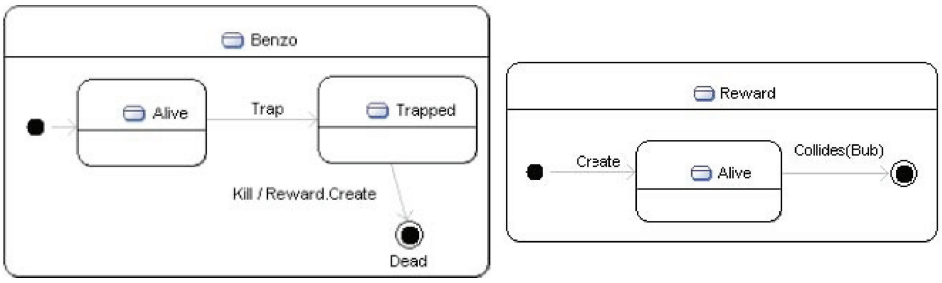

Fig. 4(c). Behavior diagrams of a Benzo enemy and a reward.

enemy inside is destroyed. If a bubble does not collide with a Benzo enemy within 10 seconds after launching, the bubble will float empty for another 50 seconds before popping by itself.

Benzo enemies (Figure 4(c)) can be in three different states: alive, trapped inside a bubble, or killed. Benzo enemies can only be destroyed when they are trapped inside a bubble, creating a reward that the dragon Bub can pick up to get extra points.

3.3 Control Mapping

By defining control mapping precisely, we can separate the game behavior from the controllers used to interact with the game (joystick, keyboard, etc.). This way, players can interact with the game through different controllers or mappings, and it will respond seamlessly. Since a control mapping diagram (Figure 5) associates a game action whit a control signal sent from a controller, the control mapping can translate control signals into game actions.

In the *Bubble Bobble* example prototype (Figure 6), the control mapping indicates that the game actions are move left, move right, jump, and blow a

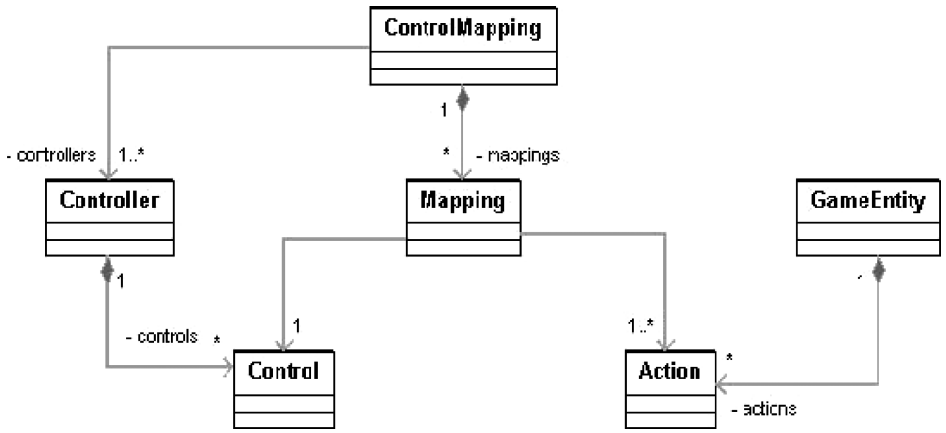

Fig. 5. Control mapping diagram.

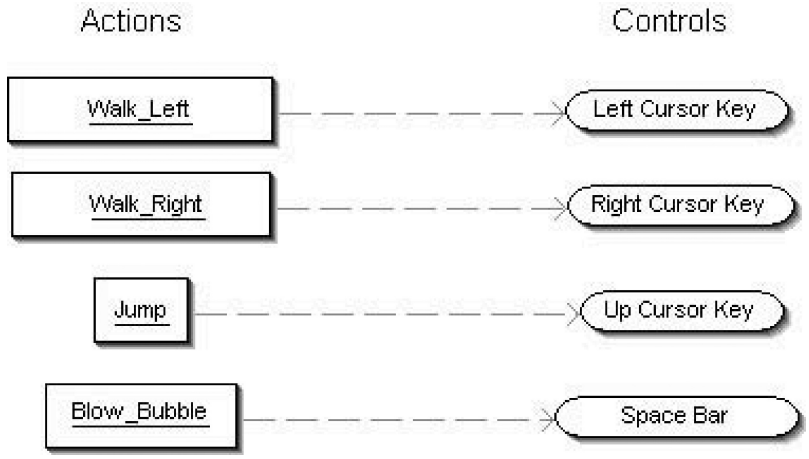

Fig. 6. Control mapping of the *Bubble Bobble* prototype.

bubble. These actions are associated to the keyboard control signals: left cursor key, right cursor key, up cursor keys, and space bar.

3.4 Transformations

The automatic generation of C++ code from models is implemented with the MOFscript transformation language.

The structure transformation creates a file (game.cpp) that implements the main loop of the game. Instances and bounding boxes for gravity and collision detection are created for each game entity that is defined in the structure diagram. The control signals described in the control mapping are checked once per frame in order to detect user actions. When a user action is detected, the corresponding event is triggered, the game state is refreshed, and the game entities are rendered on the screen.

Similarly, the behavior transformation creates a file (events.h) that implements the behavior of the game entities with C++ events. First, the events that affect the game entities are created, including the transitions of the behavior diagram and the player actions described in the control mapping. Then, a handler is defined and assigned to each event in order to implement its behavior.

4. CONCLUSION

We compared the time required to manually and automatically implement the *Bubble Bobble* example prototype. The manual implementation took a week. The equivalent automatic implementation required only a few hours, generating 93% of the game prototype (485 of 495 code lines of structure and 232 of 274 code lines of behavior). A second 2D platform game prototype example was developed: *Super Mario Bros*. All models (Sections 3.1 and 3.2) were reused in the process. The generated code represents 94% of the game prototype (393 of 393 code lines of structure and 184 of 215 code lines of behavior).

The MDGD tool for semi-automatic 2D platform game prototyping shows very satisfactory results for code generation. The increase in productivity and the reduction in development time are very significant. The resulting prototype can be reused to evolve towards the final game. The programmer has only to code aspects that are not described in the models (such as the artificial intelligence of the enemies). Furthermore, to simplify the instantiation of game entities, a graphic level editor was developed.

The development of graphic editors for behavior diagrams and control mapping diagrams remain as future work. These editors will facilitate the modeling task by providing a friendly interface for the MDGD iterative process. A further step concerning code generation can also be done by applying the Object Management Group's standard model-driven architecture (MDA) to game development. A platform-independent model (PIM) will specify a game. One or more platform-specific models (PSM) will describe how the PIM model is implemented in each target middleware platform. The transformation of the PIM model into the PSM models before code generation will separate the functionality and technology levels, which will facilitate continuous adaptation to new technology platforms.

REFERENCES

BLOW, J. 2004. Game development: Harder than you think. *ACM Queue 1*, 10, (Feb.).

CHURCH, D. 1999. Formal abstract design tools. Gamasutra. http://www.gamasutra.com/features/19990716/design_tools_01.htm.

FOLMER, E. 2007. Component-based game development – A solution to escalating costs and expanding deadlines? In *Component-Based Software Engineering*, Springer Verlag, Berlin, 66–73.

FULLERTON, T., SWAIN, C., AND HOFFMAN, S. 2004. *Game Design Workshop: Designing, Prototyping, and Playtesting Games*, CMP Books.

FURTADO, A. W. B. AND SANTOS, A. L. M. 2006. Using domain-specific modeling towards computer games development industrialization. Federal University of Pernambuco, Informatics Center, Brazil.

HENDERSON, J. 2006. The paper chase: Saving money via paper prototyping. Gamasutra. http://www.gamasutra.com/features/20060508/henderson_01.shtml.

PLEUß, A. AND HUßMANN, H. 2007. Integrating authoring tools into model-driven development of interactive multimedia applications. (Preliminary version), University of Munich.

SALEN, K. AND ZIMMERMAN, E. 2004. *Rules of Play: Game Design Fundamentals*, MIT Press, Cambridge, MA.

WAUGH, E. 2006. GDC: Spore: Preproduction through prototyping. Gamasutra. http://www.gamasutra.com/features/20060329/waugh_01.shtml.

Received November 2008; revised March 2009; accepted March 2009