Model-driven Game Development: A Literature Review

MENG ZHU and ALF INGE WANG, Norwegian University of Science and Technology, Norway

Model-driven game development (MDGD) introduces model-driven methodology to the computer game domain, shifting the focus of game development from coding to modeling to make game development faster and easier. The research on MDGD is concerned with both the general model-driven software development methodology and the particular characteristics of game development. People in the MDGD community have proposed several approaches in the past decades, addressing both the technology and the development process in the context of MDGD. This article presents the state-of-art of MDGD research based on a literature review of 26 approaches in the field. The review is structured around five perspectives: target game domains, domain frameworks, modelling languages, tooling, and evaluation methods. The article also includes reflections and a discussion of the challenges within MDGD.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; **Application specific development environments**; • **Applied computing** → *Computer games*;

Additional Key Words and Phrases: Model-driven software development, game development

ACM Reference format:
Meng Zhu and Alf Inge Wang. 2019. Model-driven Game Development: A Literature Review. *ACM Comput. Surv.* 52, 6, Article 123 (November 2019), 32 pages.
https://doi.org/10.1145/3365000

1 INTRODUCTION

Game development has progressed massively over many areas from its initial start to where we are today. However, game development is still complicated due to the following reasons [1]:

- *The requirements of computer games keep evolving throughout the development process:* Although the overall goal of game development being "entertaining" or "fun" is unchangeable, the concrete functional requirements and the game design are always subject to change during the whole development lifecycle. This is due to the gap between the anticipated gaming experience and the actual gaming experience, which cannot be guaranteed even by the most experienced game designers.
- Games have to be developed through a creative process, which makes it challenging to follow general software engineering practices directly: In terms of its artistic nature, the development team consists of a majority of people from a non-technical background. Artists work through creative processes, which are difficult to align with structured and defined

Authors' addresses: M. Zhu, Dyretraaket 26, Oslo 1251, Norway; email: mengzhu331@outlook.com; A. I. Wang, Department of Computer Science, Norwegian University of Science and Technology, Sem Sælands vei 9, NO-7491 Trondheim, Norway; email: alf.inge.wang@ntnu.no.

0360-0300/2019/11-ART123 $15.00
https://doi.org/10.1145/3365000

engineering processes. In Reference [2], Callele et al. argued that "imposing too much structure on the creative process may be highly detrimental—constraining expression, reducing creativity, and impairing the intangibles that create an enjoyable experience for the customer."

- *Communication is essential but difficult in the game development:* The success of developing computer games requires effective communication within multi-disciplinary development teams. Prominent game designers may have a good understanding of the underlying technology and can even do some coding themselves, but this usually is not the case. In practice, game designers must describe and explain most gameplay ideas to programmers, either through verbal communication or through written documentation. Documentation is highly recommended in References [3, 4], but creating useful documents is difficult, because the technical specifications of a game can easily reach 50 pages [5], and some designers insist that game design is something one cannot write down on a piece of paper [3]. Besides, game designers and game programmers do not always "speak the same language."
- *There are challenges connected to the tools and workflow* [1]: Current commercial games are intricate pieces of software, which are very difficult to develop from scratch. Game engines that emerged in the 90s have made the game development more accessible and solved parts of this problem. However, as it was pointed out in Reference [6], when the technology evolves, the demands for software also increase.

Due to the challenging nature of game development, it has a high probability of failure. Moreover, the capital investment must be considered, which can be as high as a Hollywood movie for a AAA game title. This means that the development of a massive modern game is financially risky. Failure of a big-budget game can, in the worst-case bring down major game companies. The challenges mentioned above, and the financial risk of the game industry has motivated the research on new technologies and methodologies that can ease the game development and lower the risk of game production. Model-driven Development (MDD) is a promising direction, which has shown improvements in software productivity in various software domains. Walt Scacchi also pointed out in Reference [7] that domain-specific languages that allow designers to be creative and expressive are essential for future game development. Moreover, game as a software is particularly appropriate for MDD, because:

- *MDD makes prototyping of design ideas easier:* The requirements of computer games come from the game design. Game design is a creative process where many ideas are generated, tested, evolved, and possibly thrown away, while only a few ideas are implemented in the finished product. The process of game design is iterative, or looping [4], and the iterative design emphasizes play-testing and prototyping [10]. It is also advocated that prototyping of gameplay before adding some detailed artifacts is a good practice [3]. However, implementing prototypes for validating designer's ideas is a stressful, duplicating and frustrating process for the programmers, as they have to implement changing ideas on a tight schedule, and the entire code may be dropped if the ideas are eventually considered non-entertaining. The programmers have to implement playable prototypes until the game design becomes stable continuously, which is so time-consuming that the dedicated phase called *preproduction* [5] has been introduced to the game development process. MDD can ease the preproduction phase, because, first, it allows the game designers to model and evaluate their ideas without going through a designer-programmer-designer cycle that overburdens the workflow and, second, it can save the time on manually coding the prototypes.
- *MDD reduces the requirement of developers' knowledge and experience:* A factor that makes game development difficult is that computer games have some highly domain-specific

requirements, which involves significant domain-specific knowledge [1], and this generates a huge demand for experienced programmers. When MDD is applied, proficient developers can work on the technical infrastructure such as domain-specific languages and tools, while junior developers and game designers can work on the game models that can be transformed into executable game code. Another advantage of the finer granularity of work decomposition is that the developers can dedicate themselves to a specific development domain and thus be more proficient in it.

- *MDD enables more efficient communication:* MDD's opportunity here is that models, especially models created with domain-specific languages, are formal specification of ideas. Due to the formality, communication via models can eliminate some misunderstandings of natural languages. Moreover, Domain-Specific Models (DSMs) can ease the communication between software developers and domain experts [8], in general, and we also believe it is valid for the game development domain.

When MDD is introduced to game development, Model-driven Game Development (MDGD) emerges as a new research field. Many academic studies within this field have been published during the last fifteen years, such as References [11–14]. This article presents a literature review of the MDGD approaches published up to and including 2018 and analyzes the approaches from five perspectives: (1) target game domains [45], (2) domain frameworks and target environments [48], (3) modelling languages [6], (4) tooling [60], and (5) evaluation methods. The review gives an overview of what and how MDD technologies have been used in game development without going into the technical details of each MDGD approach.

Since the MDGD field is relatively new, there are not many related works in this area. Reference [70] is a survey of MDGD for educational games. Our article differs from Reference [70] in at least three aspects:

- Our review aims at games in general, while Reference [70] focuses on the educational games domain;
- We use a different review framework from Reference [70], thus we provide different views for MDGD;
- Our review includes newer studies, as Reference [70] was published in 2011.

Reference [16] is another related work. It discusses the modeling approaches for a sub-domain of Reference [70], which is educational *Adventure* games. Since the focused game domain is narrow, there are only three approaches reviewed. Analysis and discussion of the three approaches mainly focus on the modeling language aspect. Compared to Reference [16], our review has a broader scope and uses a more systematic framework. The sources in our review are also newer.

The rest of the article is organized as follows: Section 2 presents the methodology of the review; Section 3 provides an overview of the reviewed approaches; Sections 4 to 8 analyze the MDGD approaches from the five abovementioned perspectives, respectively; and Section 9 concludes the article.

2 RESEARCH METHOD

The study presented in this article tailored the systematic review method presented in Reference [15], which uses five stages to organize the literature review: (1) "development of review protocol," (2) "identification of inclusion and exclusion criteria," (3) "search for relevant studies," (4) "critical appraisal," (5) "data extraction," and (6) "synthesis." Because the number of relevant sources found in literature was limited, and we decided to include all the qualified sources, stage 4 "critical appraisal" was skipped.

2.1 Review Protocol

A review protocol was developed to achieve the following goals:

- to maximize the literature coverage;
- to identify and include the related work where models are central artifacts in the development process;
- to find appropriate perspectives to collect and synthesize the meaningful data from the sources.

To meet the first goal, we used a two-round search to find the relevant sources. In the first round, we searched online databases to find an initial set of sources, and in the second round we went through the bibliography of the found sources and manually looked for the relevant sources that were not included in the initial set.

We realized that there are different understandings of "model-driven" in the modeling community, and in this study, we only considered the work where models are central artifacts of development, which means that the models are either executable or they can be transformed into executable code. This principle of the review was reflected in the inclusion and exclusion criteria to meet the second goal of the review protocol.

An MDGD approach can be studied from various perspectives. To meet the third goal of the review protocol, five perspectives constituting a framework were chosen to structure our review (described in Section 2.4).

2.2 Inclusion and Exclusion Criteria

Our primary purpose was to analyze existing MDGD approaches and reveal future research directions, so the included sources must document at least one MDGD approach with sufficient details for studying. Also, the approach should use the models as the central artifacts in game development, and the models must be described in a formal modeling language whose syntax and semantics are explicitly defined. The language is not only intended for a game design specification but also executable (or executable after transformation), and a tool-chain must support it. The included sources do not have to present details in all five perspectives, but they must cover sufficient details for at least one of the five perspectives. The review included articles up to and including 2018. Only works written in English were included.

Sources were excluded if the models in the approach can neither be executed, nor be transformed into executable game code. To keep the review focused, we excluded some work on conceptual models that were not intended for execution. For example, Reference [71] presented a conceptual framework and a GUI tool for creating game design documents for educational games. Since it was not about executable models, it was excluded, although it was a promising game modeling approach. Moreover, the review is about *game* development methodology, so sources were excluded if the approach documented lacked game-oriented characteristics, i.e., the approach did not mainly target the game domain. A common phenomenon is that an approach was presented in several publications with the focuses on distinct aspects of the approach, respectively. In such cases, all the sources were included and integrated as one approach during the review.

2.3 Search for Relevant Studies

The search for relevant studies was done in two phases. In the first phase, the authors searched electronic databases, and found an initial set of sources. In the second phase, the authors went through the bibliography of the initial set and identified a secondary set of sources. In the first phase, the following three electronic databases were searched:

- ACM Digital Library
- Compendex
- IEEE Xplore

We used the following keywords: (*Game* OR *Gaming*) AND *Model*.

The selection of keywords is a tricky problem: since MDGD is an emerging field, there has not been a commonly adopted terminology that can be used to precisely refer to the relevant studies. To avoid missing sources, the general keywords above were used to do the search. The downside is that it resulted in a lot of "hits" that manually had to be filtered out as they were not relevant. The percentage of relevant studies in the search results was 0.59%. Due to the high number of hits in our search, we did not search for additional general terms like "gamification" and "entertainment" to make the size of the result set manageable. However, the trade-off can potentially reduce the literature coverage of our review. To compensate, we introduced the second round of search (by bibliography).

The keywords were used in searching the titles, abstracts and indexes of the documents in the databases, and 5,933 "hits" were found. Then the authors manually excluded irrelevant and duplicated results by going through the abstract and other attributes of the publications. Most of the results were excluded in this step, and only 47 publications were considered relevant for the study. The authors read the full-texts of all 47 publications and excluded further articles using the inclusion and exclusion criteria. Eventually, 20 publications were included in the review as the results of the first searching phase. In the second phase, the authors went through the bibliography of the publications collected from the first phase and identified another 15 relevant publications.

The two-phases search eventually resulted in 35 publications, which represented 26 individual MDGD approaches, and then the full-texts of all the sources were read and analyzed by the authors. Figure 1 shows the process of literature searching.

2.4 Data Extraction and Synthesis

We investigated all the sources and extracted data for the five review perspectives:

- *Target Game Domains* [45]: The target domain describes what kind of software the MDD approach is intended to support. Computer games as a software domain is too broad for an MDD approach. All of the reviewed MDGD approaches have chosen a narrowed scope as the target domain, such as Adventure Games or Role-Playing Games. Section 4 discusses how MDGD approaches define their target domains and what these domains are.
- *Domain Frameworks and Target Environments* [48]: In MDD, models should be executable, either directly by a model interpreter or indirectly by transforming into an executable format for a target environment. In both cases, there is an underlying execution framework. Section 5 discusses the software chosen as the execution frameworks by the reviewed approaches.
- *Modelling Languages* [6]: The Modeling language is the central and fundamental artifact of an MDGD approach. Section 6 discusses the significant attributes of the modelling languages invented or adopted by the MDGD approaches, such as the syntax and semantics of the languages.
- *Tooling* [60]: The modelling languages have to be supported by a set of tools such as model editors and code generators. Tools enable MDGD in practice. Section 7 analyzes both the tools provided by the MDGD approaches and the technologies the tools are based on.
- *Evaluation*: All of the reviewed MDGD approaches have been evaluated in some way. The purposes of the evaluation include utility, usability, and productivity improvements. Different evaluation methods were chosen by the MDGD approaches such as experiments,

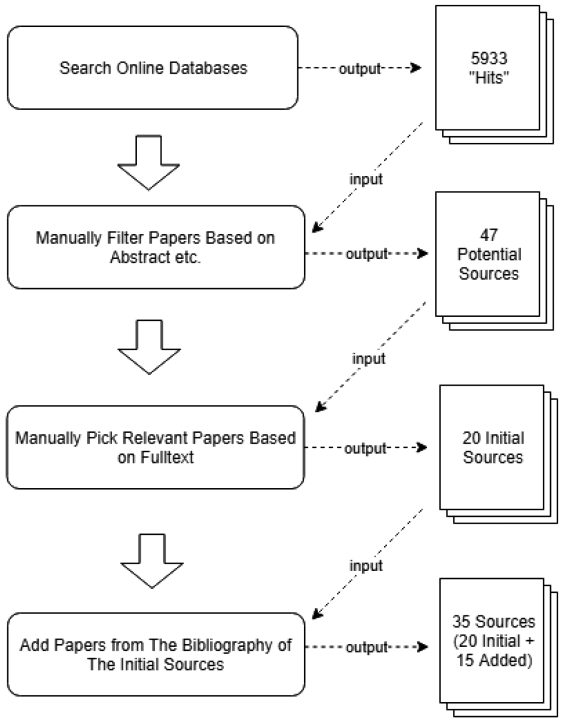

Fig. 1. Process of searching for sources.

prototypes, and case studies. Section 8 discusses the evaluation methods that were used by the approaches, and the corresponding purposes of the evaluations.

Moreover, we have identified some gaps and synthesized our reflections from the extracted data that are presented in the summary of each review section.

3 AN OVERVIEW OF MDGD APPROACHES FOUND IN THE LITERATURE

The game industry has a long tradition of using models. One example is the level models created with a level editor, which provides a visual interactive environment for game world modeling. However, the game elements the engine tools can model are restricted to a narrow scope of the game software, which is mainly assets such as game world layout and artistic data. Other elements such as AI, control, and game flow are still hand-coded, either with a native programming language such as C++ or C# or with special-purpose game scripting languages. Some state-of-art game engines do provide visual modeling tools for creating script code, e.g., the visual scripting tool in Unreal game engine. However, these tools are not taking full advantages of MDD, such as the use of meta-models and language workbenches, which makes them difficult to be adapted to new game domains. Further, the MDGD research community has explored the applications of modern MDD in the game field for years, and the results are promising.

There are mainly two categories of contributions from the MDGD research community: *MDGD Technology* and *Development Process*. Table 1 gives an overview of MDGD approaches found in the literature classified according to these two categories. A structured description is also provided for each MDGD approach, which includes the following aspects:

Table 1. MDGD Approaches Found in the Literature

Category	Description	Source
MDGD Technology	**Authors:** Sánchez et al. **Major Contributions:** A DSL and Tools **Game Genres:** Tower Defense Games **Technology Bases:** XText **Highlights:** Game models can be transformed into an intermediate language, and then transformed into code in various programming languages	[17]
	Authors: Prado and Lucredio **Major Contributions:** A set of DSLs and Tools **Game Genres:** 3D Games **Technology Bases:** XML **Highlights:** (1) Various game elements can be modeled with dedicated DSLs; (2) generated code can be integrated with manual written code	[18]
	Authors: Reyno et al. **Major Contributions:** UML tools **Game Genres:** 2D Platformers **Technology Bases:** UML, MOFscript **Highlights:** (1) Uses standardized UML technology; (2) generated C++ code can work together with manual written code	[19, 20]
	Authors: Guana et al. **Major Contributions:** A DSL (PhyDSL-2) and tools **Game Genres:** 2D Physics-based Games **Technology Bases:** Eclipse, Atlas Transformation Language (ATL) and Acceleo **Highlights:** (1) Code generator can generate code for Android; (2) DSL supports various game elements such as game objects, scene layout, and rules.	[21, 22]
	Authors: Thillainathan et al. **Major Contributions:** A DSL (GliSMo) and tools **Game Genres:** Serious Games, Point & Click Adventure **Technology Bases:** Unity **Highlights:** (1) Tools are integrated with Unity, a game engine; (2) the DSL can model both structural and behavioral aspects; (3) the tools include a model validator	[23, 24]
	Authors: Van Hoecke et al. **Major Contributions:** A DSL (ATTAC-L) and tools **Game Genres:** Serious Games **Technology Bases:** XML **Highlights:** A model interpreter is introduced to make the model executable	[25–27]
	Authors: Cutumisu et al. **Major Contributions:** A visual programming tool **Game Genres:** Role-Playing Games (RPGs) **Technology Bases:** Java, NeverWinter Nights (NWN) Engine **Highlights:** The tool was released to the NWN community and downloaded 6,000 times in 6 months. The reaction from the community has been positive.	[28, 29]
	Authors: Vargas et al. **Major Contributions:** A DSL and tools **Game Genres:** Maze Games **Technology Bases:** Model-driven Architecture (MDA), XML **Highlights:** It uses Object Management Group's (OMG) standard Model-driven Architecture (MDA)	[30]
	Authors: Minovic et al. **Major Contributions:** A DSL and tools **Game Genres:** Educational Games **Technology Bases:** XML, MDA **Highlights:** It uses MDA and only supports modeling of user interfaces	[31]
	Authors: Matallaoui et al. **Major Contributions:** A DSL (GaML) and tools **Game Genres:** Serious Games **Technology Bases:** Unity, XText **Highlights:** It supports a popular game engine: Unity	[32]

(Continued)

Table 1. Continued

Category	Description	Source
	Authors: Pleuss et al. **Major Contributions:** A set of DSLs (MML) and tools **Game Genres:** Flash Games **Technology Bases:** Eclipse Modeling Framework (EMF) **Highlights:** It supports both behavioral and structural modeling of games with dedicated DSLs	[33, 34]
	Authors: Marques et al. **Major Contributions:** A DSL and tools **Game Genres:** Mobile RPGs **Technology Bases:** Eclipse Modeling Framework (EMF) **Highlights:** (1) Code generation is implemented through a two-step process including a model-model transformation, and a template-based model-code generation; (2) ALPiNA [35]-based semantic validation is also implemented	[36]
	Authors: Stürner and Brune **Major Contributions:** A DSL and tools **Game Genres:** Virtual Worlds **Technology Bases:** EMF **Highlights:** It can generate JavaScript code for web browser	[37]
	Authors: Marchiori et al. **Major Contributions:** A DSL and tools **Game Genres:** Educational Games **Technology Bases:** Mealy Finite State Machine **Highlights:** The DSL has a formal theoretical base and supports domain specific features	[38]
	Authors: Valdez et al. **Major Contributions:** A graphical modeling toolset (Gade4all) **Game Genres:** Various genres such as puzzle, platformer. **Technology Bases:** XML **Highlights:** Toolset supports multiple game genres and can generate code for various platforms such as iOS, Android, HTML5	[39]
	Authors: Morales et al. **Major Contributions:** A Model-Transformation Chain (MTC) **Game Genres:** Maze Games **Technology Bases:** MDA and EMF **Highlights:** The MTC proposed includes three steps: (1) transformation from game Platform-Independent Models (PIMs) to architecture PIMs; (2) transformation from architecture PIMs to game Platform-Specific Models (PSMs); and (3) generate code (JavaSE or Torque 2D) from game PSMs	[40]
	Authors: Funk and Rauterberg **Major Contributions:** A DSL (PULP) and tools **Game Genres:** HTML5 Games **Technology Bases:** XText **Highlights:** PULP supports both structural modeling and behavioral modeling of games	[41]
	Authors: Furtado and Santos **Major Contributions:** Two DSLs (SLGML and Head-Up Display Creation DSL) and Tools **Game Genres:** 2D Adventure Games **Technology Bases:** Microsoft Visual Studio DSL Tools **Highlights:** The approach follows Software Factory engineering methodology	[42]
	Authors: Hernandez and Ortega **Major Contributions:** A DSL (Eberos GML2D) and Tools **Game Genres:** 2D Games **Technology Bases:** Microsoft Visual Studio DSL Tools **Highlights:** Eberos GML2D supports both structural and behavioral modeling of games	[43]

(Continued)

Table 1. Continued

Category	Description	Source
	Authors: Zhu et al. **Major Contributions:** A modeling approach (ECGM) and A DSL (RAIL) **Game Genres:** Action/Adventure Games **Technology Bases:** EMF and Acceleo **Highlights:** The ECGM approach bridges engine-based game development and model-driven game development, enabling the cooperation of tools from these two development methods.	[72, 73]
	Authors: Ferreira et al. **Major Contributions:** An XML-based DSL (LEGaL) for creating Location-based Games **Game Genres:** Location-based Games **Technology Bases:** XML and Nested Context Language (NCL) **Highlights:** NCL is a declarative language for hypermedia documents authoring, and LEGaL extended NCL to support modeling and generating Location-based Games.	[74]
	Authors: Aouadi et al. **Major Contributions:** A DSL and tools for creating serious games **Game Genres:** Serious Games **Technology Bases:** MDA and EMF **Highlights:** It has implemented a DSL and tools following MDA standard, which can generate serious game code for the Unity engine.	[75]
Development Process	**Authors:** Maier and Volk **Major Contributions:** An MDGD process **Game Genres:** Pac-Man-like Games **Technology Bases:** EMF **Highlights:** The process follows Software Factory engineering methodology and emphasizes the use of language workbenches	[44]
	Authors: Walter and Masuch **Major Contributions:** An MDGD process **Game Genres:** 2D Point and Click Adventure Games **Technology Bases:** XPand and XText **Highlights:** The process follows a bottom-up approach starting from language requirements to reference artifacts, then up to language definition and finally the domain-specific programs	[14]
	Authors: Furtado et al. **Major Contributions:** An MDGD process **Game Genres:** 2D Arcade Games **Technology Bases:** Microsoft Visual Studio DSL Tools **Highlights:** The process is iterative, and it combines both top-down and bottom-up methods, where the elaboration of the problem domain and the solution domain is done alternately	[45, 46]
Both Technology and Process	**Authors:** Guo et al. **Major Contributions:** An MDGD process, a DSL and tools **Game Genres:** Pervasive Games **Technology Bases:** EMF **Highlights:** (1) It uses ontology-based domain analysis and iterative process to ensure effective MDD workflow; (2) a textual DSL is proposed for modeling pervasive games	[47]

- Authors of the proposed and published approach;
- Major contributions of the MDGD approach;
- Game genre of the approach targeted, or game genre used for validation;
- Technology Bases being the technical standards or frameworks is the approach based on;
- Noteworthy features of the approach.

Note that some approaches have more than one publication, so to distinguish each approach, we format its citations specially in this article as *[number, number…]*.

Table 1 can be used as the entry point for searching for information of interest in this article: Readers can first choose which category (technology or process) of existing studies they are interested in, and then go through that category in Table 1 to get a quick glance of the approaches through the structured description. If the reader needs more detailed information about:

- a specific approach, then he or she can find the sources for the approach in Table 1; or
- a specific aspect of MDGD, then he or she can go to the corresponding review section among Sections 4–8 to get the details of that aspect, and the comparison results across the approaches as well.

4 TARGET GAME DOMAINS

An essential value of MDD is to raise the abstraction level, which allows specifying a solution with problem domain concepts, thus making the solution more straightforward and more accessible. To this end, both the modeling language and the code generator should be domain-specific [48]. Choosing a proper target domain is essential to the success of an MDGD approach. If the target domain is too narrow, then the application of the approach will be restricted. Therefore, it will be challenging to pay off the initial investment. However, if the target domain is too broad, then it will result in other issues such as:

- *Unmanageable number of language constructs:* The broader the target domain is, the more concepts the modeling language has to support. A too-large number of concepts can make the modeling language too complex to develop and maintain.
- *Low abstraction level:* To control the complexity of a modeling language targeting a too broad domain, the abstraction level of the language may have to be sacrificed; therefore, the language will become less useful. In extreme cases, the abstraction level can be as low as a third-generation programming language (such as C or C++), which eliminates the advantages of MDD.
- *Poor descriptiveness:* If the modeling language targeting a too broad domain needs to keep both high abstraction level and manageable complexity, then it has to sacrifice the descriptiveness by supporting only a minimal set of concepts in the domain. The language capability is thus limited, and the language may need a continuous update.

The rest of this section presents how the reviewed MDGD approaches define their target domains, and a discussion of the remaining challenges.

4.1 Genre-based Domains

The reviewed MDGD approaches have different methods for defining the target domain, where defining the domain based on a game genre is the most common choice. Game genres are categories of games based on features of the gameplay. Note that "genre" is an informal categorization, and genres can have ambiguous definitions.

References [19, 20] target "2D Platform Games (Platformers)," which is a kind of action game featuring the player character jumping between various platforms. References [72, 73] target "Action/Adventure Games," which is also action games but can support more complex player and AI behaviors than platform games do. Reference [17] supports "Tower Defense Games", whose gameplay is about building static defensive units to protect your base from being captured by incoming enemies. Two MDGD approaches are targeting "Role Playing Games (RPGs)": References [28, 29] and [36]. RPG is a popular game genre, where players act as a character in an interactive virtual world, which usually has a compelling background story. References [28, 29]

support RPGs targeted for the Neverwinter Nights'[1] engine, and Reference [36] targets RPGs on mobile platforms. "Maze Game" is a kind of top-viewed 2D games played on a gridded map. Each cell of the map can be an obstacle or open ground. Enemies and player characters move on the map and conflicts may occur. References [30] and [40] both target the Maze Games genre. Reference [14] uses "Point & Click Adventure Games (ADVs)" as the target domain, which are games featuring pointing and clicking on graphical objects rendered on a static background and players interacting with Non-Player Characters (NPCs) to collect clues to unfold a story. References [23, 24] support ADVs, but only focuses on the ADVs for educational purposes. References [21, 22] supports "2D Physics-based Games," which is a kind of 2D action games building gameplay around physics simulation. Reference [39] is different from the above approaches, as it provides an integrated tooling environment, Gade4all, to support multiple genres, such as Platformers and Puzzle Games. Gade4all can also be extended to other game genres.

"Educational Games" are games that are created for teaching or learning purposes. Educational Games as a genre is controversial, because its gameplay can hardly be distinguished from other genres. For example, an educational game can be an RPG, an ADV or a Puzzle Game. "Serious Games" is a superset of Educational Games that include not only educational games but also other games with serious purposes, such as games for physical exercise and games for health. There are quite a few MDGD approaches targeting serious games or educational games, such as References [23, 24], [25–27], [31], [32], [38], and [75].

Choosing an existing game genre can easily reuse the shared domain knowledge and making the domain definition intuitive and straightforward. However, game genres do not have formal definitions. Therefore, they cannot precisely define the application scopes of the MDGD approaches. However, a well-defined application scope is essential for developing and applying such an approach. The drawbacks of genre-based domains are discussed in detail in Section 4.5.

4.2 Systematically Defined Domains

Several MDGD approaches use systematic methods to define the target domain. Reference [42] outlines the target domain as "Adventure Games," and then provides a detailed description from nine perspectives corresponding to nine key features of the game domain. In References [45, 46], ArcadEx goes further through using a feature model in addition to the keyword "2D Arcade Game" to characterize the target domain. The model includes 150 features describing the domain's commonality and variability. Similarly, Reference [47] targets "Pervasive Games," which is a game genre where the gameplay integrates the virtual and physical world. Reference [47] uses a Pervasive Games ontology, "PerGO" [49], to systematically analyze the commonality and variation of the target domain. Location-based Games are a sub-type of Pervasive Games, which feature the utilization of players' physical position in creation gameplay. In Reference [74], the features and four major gameplay patterns of Location-based Games are summarized in terms of a survey and analysis of 15 Location-based Games in literature, which constitute its domain definition.

These systematic methods produce more formal and useful domain definitions, which provide well-defined goals for the MDGD approaches development and help the users to understand the approaches easily.

[1]Neverwinter Nights is originally an RPG released by BioWare, and due to its extensibility, many mods are created. Today an active community has created over 4000 mods for Neverwinter Nights and it has become a development platform for custom RPGs. The community website: www.neverwinternights.info.

4.3 Single Game Domains

Two approaches define extremely narrow target domains: a single game or a game family consisted of a couple of very similar games. Reference [44] presents MDD for Pac-Man like games, which follows a "one game at a time" principle: For a particular game or game family, a complete MDD tool-set is created, which is optimized for the development of the project and can hardly be reused without significant modifications in other projects. Creating DSLs and tools for one game may be economically unfeasible, so the language workbench is promoted as the core for the approach, which can significantly save the cost of MDGD tooling. References [25–27] are more examples of creating MDD tools and languages for a single game project, and the target game is a serious game for preventing cyberbullying.

4.4 Other Domains

Some MDGD approaches choose domains broader than the genre-based domains. Reference [37] intends to facilitate the development of "Virtual Worlds" by visualizing the construction of scenes and characters. Virtual Worlds include games but are not restricted to games, so the domain has a very broad scope. Reference [41] claims to support "Interactive HTML5 Applications," which is also a vast domain. Similarly, References [33, 34] target Adobe Flash games, which is a rival technology of HTML5. These domains are so broad that each of them covers many if not all game genres, whose domain concepts can hardly be packed into a single DSL. In practice, these approaches only support a sub-set of the target domains.

Hernandez and Ortega argued that the 2D game domain is narrow enough for MDGD, and they created Eberos GML2D in Reference [43] to support the development of *all* variations of 2D games. The DSL includes generic concepts for 2D games such as sprites, entities, animations, state machine, and messages. They also validated the approach by modeling and generating two games. However, the two games demonstrated are both Platformers. It is doubtful whether the approach scales to the tremendous diversity of 2D games. Similarly, Reference [18] chose "3D games" as the target domain, and a set of DSLs were proposed to support three sub-domains, namely *camera, character* and *scenario* (game scene). The limitations of References [43] and [18] are mainly about the expressiveness of the DSLs: 2D/3D games include games ranging from board games to MMORPGs, which have enormous number of distinct domain concepts, while the DSLs have supported only a limited subset. We have to keep in mind that "developers often create a language that is too generic for its domain, with concepts and semantics that are either too few, too generic, or both," and that is one of the worst MDD practices [50].

4.5 Summary of Target Game Domains

Table 2 summarizes the target game domains of the MDGD approaches in our review: The majority of the approaches (15 of 26) choose genre-based domain definition; four approaches define their target domains with more systematical methods; two approaches use a single game or a game family as the target domains; and five approaches claim to support game domains broader than a single game genre.

We want to point out two significant problems with the *Genre-based Domains*:

- *There is a gap between the game genres claimed to be supported and the actual game domains supported:* Game genres is an informal categorization of games, which is not precisely defined, so the MDGD authors may not share the same understanding of genres as developers using their approaches. Moreover, a game genre can have many variations, which usually exceed the feasible scope of an MDGD approach in practice. For example, RPG is a quite large genre that has sub-genres such as Action RPG, Tactical RPG, Sand-Box

Table 2. Domain Definitions of The MDGD Approaches

Category	Target Domain	MDGD Approach
Genre-based Domains	2D Platformers	[19, 20]
	Action/Adventure Games	[72, 73]
	Tower Defense Games	[17]
	RPGs	[28, 29], [36]
	ADVs	[14], [23, 24]
	Maze Games	[30], [40]
	2D Physical-based Games	[21, 22]
	Multiple Genres such as Puzzles, Platformers	[39]
	Educational Games	[31], [38]
	Serious Games	[32], [23, 24], [75]
Systematically Defined Domains	ADVs with a detailed description from nine perspectives	[42]
	2D Arcade Games defined with a feature model	[45, 46]
	Pervasive Games with a detailed definition based on PerGO ontology	[47]
	Location-based Games with key features and gameplay patterns identified from analyzing 15 games in literature	[74]
Single Game Domains	Pac-Man Like Games	[44]
	Friendly ATTAC: A Serious Game for preventing cyberbullying	[25–27]
Other Domains	Virtual Worlds	[37]
	Interactive HTML 5 Applications	[41]
	Flash Games	[33, 34]
	All 2D Games	[43]
	3D Games	[18]

RPG, and Simulation RPG. The MDGD approaches using RPGs as the target domains only support a limited set of RPGs.

- *Domain concepts are not entirely supported in the prototype implementations of the MDGD approaches:* The MDGD approaches are developed in research environment with limited resources, which constrains the complexity and maturity of the prototype implementations of the languages and tools. For example, References [19, 20] target 2D platform games, and does support some important domain concepts such as entities and player characters, but many essential concepts of the game genre are still missing, such as physics, camera control, and inventory.

The above gaps make it difficult to evaluate the suitability of an MDGD approach for a given game project and therefore threatening the usefulness of the MDGD approaches. Note that the problems are even more significant with the approaches using the broader domain definitions (the *Other Domains* category in Table 2).

Use of a *Single Game Domain* solves the problems by narrowing down the scope of MDGD to a single game or a game family that is easier to define and support. Moreover, since the modeling languages and tools are dedicated to a specific game, game-specific features can be added to the tools and languages on-demand, which maximizes the gameplay flexibility, freeing the game designers from tool-restrictions.

The *Systematically Defined Domains* provide the most precise definition of the target domain, making it easier for MDGD developers to understand the requirements and for MDGD users to evaluate the suitability of the languages and tools. We found three systematic domain definition

Table 3. Domain Frameworks of Reviewed MDGD Approaches

Category	Domain Framework	MDGD Approaches
Use general computation platforms	OS	[17], [19, 20], [39], [44]
	J2ME	[31]
	Web Browser	[37], [41]
Use game engines or equivalent software	jMonkeyEngine	[18]
	Box2D Engine	[21, 22]
	Unity Engine	[23, 24], [32], [75]
	Neverwinter Nights	[28, 29]
	Adobe Flash	[33, 34]
	e-Adventure Engine	[38]
	JavaSE/Torque2D	[40]
	Microsoft XNA	[43], [14]
	LAGARTO tool (LocAtion-based Games AuthoRing TOol) [76]	[74]
Use modified game engines	Modified Corona SDK	[36]
	Modified FlatRedBall Engine	[42]
	A modified game engine, unknown name	[45, 46]
	Modified Torque 2D	[72, 73]
Use model interpreters	Self-developed Interpreter	[25–27]
N/A	Not Presented	[30], [47]

methods: (1) nine-perspectives domain description method in Reference [42], (2) feature-model-based method in References [45, 46], and (3) ontology-based method in Reference [47].

All the above methods have their pros and cons, and we believe that the choice of the most appropriate domain definition method depends on the characteristics of a specific project.

5 DOMAIN FRAMEWORKS AND TARGET ENVIRONMENTS

The models in MDD are either directly executable or able to be transformed into the executable code. For the former, a model interpreter is required to process the model semantics. For the latter, a software layer between the generated code and the underlying target environment is usually necessary [48]. The interpreter and the software layer in the middle are so-called Domain Frameworks in MDGD. For example, a game engine is a typical domain framework. Table 3 shows the domain frameworks of the reviewed MDGD approaches.

The domain frameworks used by the reviewed MDGD approaches mainly fall into four categories. The *first category* includes approaches where the generated code runs directly on top of a general platform, such as OS, web browsers and the Java platform. In such cases, the generated games are relatively simple, and due to the higher abstraction level of models, they are usually portable among various platforms. The *second category* uses game engines or equivalent software components as the domain frameworks. Game engines provide a common implementation of low-level technologies for the generated games, thus support more complex gameplay. Note that we include References [33, 34] in this category, although the domain framework it used is Adobe Flash, which is not a game engine but instead can be classified as a multimedia engine. The *third category* is modifying a game engine to promote it to a domain framework, as it was suggested in Reference [51]. Modifying a game engine can increase its abstraction level thus reduce the semantic gap between the game model and the engine application interface to make the code generation easier to implement. The *last category* is using a specific semantics engine to interpret the models in runtime. References [25–27] is the only approach in this category that has

implemented a semantics engine on top of a game engine. It can load the models and feed them to the game engine, which renders the game graphics as well as handles the player interactions.

The reviewed approaches show that choosing a domain framework is dependent on the project context: for simple games such as educational games and prototypes, OS and web browsers are proper options; for commercial games, a game engine is undoubtedly the mainstream choice, because it is too difficult to re-implement the features provided by game engines. Furthermore, in many cases, modifications to the game engine, such as adding a layer are useful. The modifications make the game engine a domain framework, including "utility code or components to make the generated code simpler" [48]. The model interpreter is an unusual solution to support the model semantics. It differs from a code generator at two stages: analysis time and mode of execution [52]. Interpreters have two distinct advantages: they avoid code generation and compiling and thus simplifies the workflow, and they also support runtime update of models. Model interpreters have been used in other software domain, such as business process modeling, and they can also be useful to commercial game development, because compiling generated code and making binary images is time-consuming [1].

6 MODELING LANGUAGES

The Modeling Language is the central part of MDGD. "A language provides the abstraction for development and as such is the most visible part for developers" [48]. This section discusses modeling languages proposed or used by the MDGD approaches. We use the framework introduced in Reference [77] to structure the discussion, which was proposed for classifying design guidelines for modeling languages and includes five perspectives: (1) *Language Purpose*, (2) *Language Realization*, (3) *Language Content*, (4) *Concrete Syntax*, and (5) *Abstract Syntax*. The framework is modified a bit as follows for our needs:

- *Language Purpose* is about what the language needs to describe (domain) and what the language can be used for (usage). For usage, general examples include *documentation, development*, and *generation tests* [76]. However, due to the scope of our review, the modeling languages used by the MDGD approaches always serve the *development* purpose, so for the Language Purpose discussion, we will only focus on the language domain aspect.
- *Language Realization* is about how to implement the language, which is an essential topic of our review. We have an individual section (Section 7) that discusses the implementations of the languages in terms of the tools they provide and the underlying technologies that enable the implementations, so we do not include the perspective again in this section.

6.1 Supported Domain Concepts

One of the first activities in language design is to analyze the aim of the language [77]. The result of the analysis includes the usage of the modeling language and the domain concepts that the language should support. We will not discuss the usage of the languages here, as this section focuses on how the MDGD approaches identify the relevant domain concepts in the language design. As it was summarized in References [48, 54], there are five popular methods for finding the relevant domain concepts for modeling languages: (1) "physical product structure," (2) "look and feel of the system," (3) "variability space," (4) "domain expert concepts," and (5) "generation output." Table 4 shows methods being used by MDGD approaches to find language constructs. The MDGD approaches that do not present such information are put in the last row in the table.

Look and Feel of the system is about how the system externalizes its states and how users interact with the system. In the computer games domain, it is mainly about the game visualization and gameplay. Unsurprisingly most of the MDGD approaches use it as the main source for language

Table 4. Sources of Language Concepts

Language Concepts From	MDGD Approaches
Physical product structure	[47]
Look and feel of the system	[17], [19, 20], [42], [14], [21, 22], [36], [40], [72, 73]
Variability space	[45, 46], [18], [47]
Domain expert concepts	[14], [23, 24], [32], [33, 34], [38], [43]
Generation output	[45, 46]
N/A	[25–27], [28, 29], [30], [31], [37], [39], [41], [44], [74], [75]

concepts. In the simplest case, the target domain is outlined with several sentences where core concepts were revealed and identified, e.g., Reference [40]. Furthermore, Reference [42] describes the target domain from nine perspectives, providing more specific information. References [21, 22] perform a systematic domain analysis by answering five questions about the look and feel of the target game domain to reveal the core concepts. Moreover, some approaches analyze the look and feel of one or more game instances in the target domain to understand the target domain and identify the core concepts, such as References [17] and [19, 20].

Variability space-based analyses are used by some approaches to find the language concepts. Reference [18] uses several prototypes to explore the possible variabilities of the domain before implementing the modeling language. Reference [47] uses an ontology named PerGO [49] to analyze the variabilities.

Reference [14] encourages the participation of *domain experts* (game designers) in the language design phase, which can provide valuable insights necessary for technical people to avoid misunderstandings of the target domain. In Reference [43], one of the authors himself is a domain expert, and the language design integrates the results from discussion with other domain experts. When domain experts are not available in person, a practical alternative may be studying literature by domain experts, which is used in a few approaches such as References [23, 24] and [32].

The *generation output* on the low abstraction level can reveal the pros and cons of the existing language. It is used by References [45, 46] as part of its most systematic domain analysis method: An iterative process that combines variability analysis and generation results analysis, where the variability analysis results in a language and analysis of the generation results provides feedback on the current language, and the language is continuously improved.

Physical product structure is only used in Reference [47], which targets pervasive games. Pervasive games generate gameplay in the intersection of the physical and the virtual world. Thus, the physical aspects of the gaming system become relevant in designing the modeling language.

6.2 Language Content

One Main activity in language development is the task of defining the different elements of the language [77]. The language elements are means to map the problem domain concepts to the solution domain. We can identify three major categories of language elements that are provided by the MDGD modeling languages: (1) Structural Elements, which support modeling the structural aspects of games, e.g., program static structure, game scene structure, game entities structure; (2) Behavioral Elements, which support modeling the behavioral aspects of games, e.g., game control, game flow, character behavior; and (3) Hybrid Elements, which serve both structural and behavioral modeling, e.g., User Interface related elements. Table 5 presents an overview of the categorized language elements provided by the MDGD languages.

Table 5. Language Elements Provided by The MDGD Modeling Languages

Elements Categories	Sub-Categories of Language Elements	Sources of The Modeling Languages
Structural Elements	Program Static Structure	[19, 20]
	Scene Structure	[17], [18], [21, 22], [23, 24], [25–27], [30], [36], [47], [37], [39], [40], [44], [14], [42], [45, 46], [74], [75]
	Game Entities	[17], [18], [21, 22], [25–27], [30], [36], [47], [37], [39], [40], [43], [14], [42], [45, 46]
	Global Configuration	[18], [21, 22], [45, 46]
Behavioral Elements	Player Character Control	[19, 20], [21, 22], [40], [45, 46]
	Game Object Behavior	[17], [19, 20], [21, 22], [23, 24], [25–27], [28, 29], [37], [40], [43], [14], [42], [72, 73]
	Game Flow	[25–27], [33, 34], [38], [42], [45, 46], [74], [75]
	Global Rules	[21, 22], [32], [41], [14]
Hybrid	User Interface/Heads-Up Display	[31], [33, 34], [39], [41]

We further elaborate the three major categories into nine sub-categories in Table 5, which are:

- *Program static structure:* It is about how the data structure and process are organized, e.g., a class view in Object-Oriented Design (OOD). This type of model is usually regarded as a meta-model in DSL-based approaches, and in General Purpose Language (GPL)-based approaches, e.g., UML, it can be supported by the GPL. Only References [19, 20] provide elements in this category, as a UML-based approach.
- *Scene Structure:* Languages supporting Scene Structure modeling should provide elements that can specify what game entities a game scene contains and the relationships between the entities. A Scene Structure model usually includes two views: a logic view representing what the scene contains and a layout view representing the spatial relationships of the content.
- *Game Entities:* It is about the individual objects in a game scene, which can be a Player Character (PC), a Non-Player Character (NPC), Gameplay Objects (GO), or similar. Elements for modeling game entities usually support the creation, removal, and configuration of game entities in game scenes.
- *Global Configuration:* It is about the high-level game settings, for example, screen resolution, camera mode, and session time, which significantly impact the gaming experience.
- *Global Rules:* It defines the high-level rules of the game, including the game-ending rule, scoring rule, or similar.
- *Game Flow:* It is about how the game unfolds and proceeds. Language elements in this category support modeling the temporal and logical connections between game scenes, which constitute an organic narrative structure.
- *Game Object Behavior:* It is about how game entities interact with each other and the player. The interaction is essential for almost all kinds of gameplay. In traditional game development, the *Game Object Behavior* has to be implemented through programming and is very resource consuming. By providing language elements for Character Behavior, MDGD can significantly reduce the development cost.
- *Player Character Control:* It is about how the user input is mapped to the player character (avatar) moves. Language elements such as the input-to-action map belongs to this sub-category.
- *User Interface (UI):* It is about the visual components used for user interaction, such as buttons and menus. A sub-set of User Interface components for displaying in-game players

Table 6. Forms of Concrete Syntax of The Modeling Languages

Forms of Concrete Syntax	MDGD Approaches
Graphical	[18], [19, 20], [23, 24], [33, 34], [36], [38], [39], [43], [44], [45, 46], [42], [75], [25–27]
Textual	[17], [30], [31], [32], [47], [41], [21, 22], [44], [14], [74]
Tree-View	[28, 29], [37], [72, 73]
Form-based View	[18], [39]
N/A	[40]

information such as scores and lives is called *Heads-Up Display* (HUD) in the game community. UI is a large domain, so some languages chose it as a major, or even the sole target of the language elements.

6.3 Concrete Syntax

The concrete syntax, i.e., the language representation, determines the first-sight impression of the language. "A poorly chosen concrete syntax will drive users away, stopping them from using even the most wonderful language" [50]. Table 6 shows the forms of concrete syntax chosen by the MDGD approaches.

The most commonly used forms of concrete syntax are the graphical and the textual forms. Graphical representation is usually more intuitive. By using graphical representation, the language notation and its symbols can directly represent domain concepts, which is the best practice for selecting symbols [48]. However, graphical languages does not always over-performs textual languages, as discussed in Reference [55]. Another study also revealed that "higher proportion of developers might be predisposed to choose text given its traditional prevalence in programming" [50]. To take the advantage of both forms, Engelen and Brand explored the potential of integrating graphical and textual representations [56]. We agree that a combination of graphical and textual representations can improve the usability of game DSLs.

Tree-view is a special kind of graphical form, which is good at presenting a hierarchical structure. In game scene modeling, it is particularly useful, because a game scene can usually be organized in a tree structure. Furthermore, when using Eclipse Modeling Framework (EMF) as the tooling environment, you can get a tree-view model editor with negligible development effort, which is very attractive for researchers who need proof-of-concept implementations. A form-based view allows setting attributes of modeling objects by filling a form with a group of fields, where each field must be filled with a specific type of value. The form-based view is convenient in modeling attributes of game entities [39], and global configuration [18].

6.4 Abstract Syntax

The abstract syntax is a model of the proper sentences of the language [78], which defines the concepts, relationships, and integrity constraints available in the language [9]. The abstract syntax of a modeling language is usually defined through meta-modeling [53], while every model created by the modeling language is an instance of the language meta-model. Abstract syntax determines what a modeling language can describe and how.

For the reviewed MDGD approaches, the abstract syntaxes of their modeling languages vary; thus, the forms of supported game models are different as well. In Table 7, we classify the abstract syntaxes of the modeling languages based on what kinds of models they can support. Note that each kind of abstract syntax has different domain appropriateness. Therefore, one modelling

Table 7. Classification of The Modeling Language Abstract Syntaxes

Abstract Syntax Categories	Source MDGD Approaches
Code-Like models	[21, 22], [25–27], [47], [41], [28, 29]
Game Object Specification	[18], [23, 24], [44], [42], [17], [30], [31], [14], [74], [36], [37]
Directional Graphs	[19, 20], [33, 34], [36], [43], [75]
State Machine Models	[17], [19, 20], [23, 24], [33, 34], [38], [43], [72, 73]
Production Rule Systems	[19, 20], [42]
Decision Trees	[14]
Graph Transformation	[44]

language may use multiple kinds of abstract syntaxes to support modeling various game aspects appropriately.

Mainstream programming languages follow the *imperative* programming paradigm that defines computation through a sequence of steps [8]. The first category of abstract syntax in Table 7 supports models following the same paradigm that define game behaviors by describing the detailed control flows. We name this category "*Code-like*" models, because they are similar to programs from the abstract syntax perspective. Note that although the models have code-like nature, the abstract syntaxes are on a higher abstraction level than the programming languages, and they do not have to be externalized by a textual concrete syntax, e.g., References [25–27] (graphical) and References [28, 29] (tree view).

The second category in Table 7 supports the *specification of game objects*, which commonly includes spatial data, presentation data, and game-specific data of game objects. The game objects are usually organized with a hierarchical structure (scene tree) based on their logical relationships such as containment, ownership, and grouping. In the traditional game development method, such models are created by a World Editor, which is a component of game engines. Quite a few modeling languages are based on such abstract syntaxes, but they often include additional syntax to improve the expressiveness on behavioral modeling.

A *Directional Graph* consists of two sets: a node set and an edge set, where edges connect nodes, and each edge has two ends with different meanings. A typical Directional Graph is the UML class diagram, which is used in References [19, 20] to model the static structure of the game software. Other examples include References [33, 34], which use a UML-like notation to model user interface elements and corresponding media objects, and Reference [36], which uses Directional Graph to model game scene, to name a few.

State Machine is a standard tool for modeling processes or object behaviors. A State Machine consists of a set of states. For a given state, the behaviors of the machine, which may include a state-transition, are triggered with events depending on the current state [8]. State Machine is supported by References [33, 34] for modeling UI interaction, and by Reference [43] for modeling entity behaviors. Reference [38] extends the State Machine by adding domain-specific features such as hierarchical structure, parallel structure, and multi-interaction node to support modeling narrative aspects of games. References [72, 73] also adapted State Machine to its target domain by adding concepts such as *Triggers* and *Actions* that are useful for modeling NPC behaviors.

A *Production Rule System* implements "the notion of a set of rules, where each rule has a condition and a consequential action" [8], and the rules are run in cycles whose actions are executed if its conditions match [8]. The modeling language used by References [19, 20] supports the Production Rule System for modeling the game control, which maps each input event (condition) to a character action. Reference [42] defines more types of conditions than input events, and the actions to be triggered are not only restricted to character actions but are extended to various game state

manipulations such as spawning game objects and playing animations. Moreover, the conditions in Reference [42] can be the combination of multiple atomic conditions in conjunction, and so can the actions.

Decision Tree [58] is a widespread technique in the information theory field. *Decision Tree* syntax is used by Reference [14] to specify interactive conversations. An interactive conversation is a tool for unfolding the story, or for determining the game flow. It usually starts from a character saying something to another, and the latter can respond by choosing one option from a set of answers, which further drives the other characters to respond, and this process repeat until the conversation ends. The process of conversation naturally conforms to the decision-tree structure.

A *Graph Transformation* can specify how models evolve, and when structural models are defined with a Graph, behavioral computations are naturally modeled as graph transformations [57]. *Graph Transformation* syntax is used by Reference [44] for modeling the game logic.

6.5 Summary of The Modeling Languages

Modeling languages as the heart of MDD has been the focus of the design and development for most of the MDGD approaches. A broad range of methodologies and technologies from MDD communities are applied. For example, to identify the relevant domain concepts for modeling languages, all five methods found in the MDD literature have been used. Moreover, in References [45, 46] multiple methods are integrated as a systematic approach that is particularly suitable for finding domain concepts.

The content of modeling languages used by the MDGD approaches has a broad coverage for game software, from scene structure to character control. We also notice that the reuse of technologies from other software domains is worth further exploration. For example, Graphical User Interfaces (GUIs) are similar for games and other graphical software, but none of the MDGD approaches have attempted to adopt existing GUI modeling technologies.

The MDGD approaches displayed appropriate use of different forms of concrete syntax, from textual to graphical, which improved the descriptiveness and readability of the modeling languages. There are still some forms of concrete syntax with potentials not being used, such as matrix and table. They may over-perform others in a specific context, e.g., "a matrix is especially good if connections between model elements are important, and a matrix also scales better than a diagram, since more information can be show in the same space" [48].

The abstract syntax of modeling languages determines the form of the game models. The modeling languages used by the MDGD approaches have various abstract syntaxes thus support diverse forms of models, such as *Game Object Specification, State Machine,* and *Graph Transformation*. The abstract syntaxes presented still have potentials to improve, for example, the chaining of the *Production Rule System*: when rule actions change the state of the software, all rule conditions need to be evaluated to find if any has become true, and the found rules will be added to the agenda for further execution [8]. More forms of models may also worth exploration, such as *Decision tables*, which "display elements of data in a way that makes it clear what combinations of values result in specific courses of action" [59].

7 TOOLING AND TOOLING ENVIRONMENT

Model-driven Software Development does not make sense without tool support [52]. Kent also advocated in Reference [60] that "Tooling is essential to maximize the benefits of having models, and to minimize the effort required to maintain them." This section discusses the tools provided by the existing MDGD approaches and the tooling environments for creating and supporting them.

Table 8. Tool Support of the MDGD Approaches

Tool Support	MDGD Approach
Code Generator	[19, 20], [21, 22], [30], [31], [40], [74]
Model Editor and Code Generator	[17], [18], [28, 29], [32], [33, 34], [37], [38], [39], [41], [43], [14], [45, 46], [72, 73], [75]
Model Editor and Interpreter	[25–27]
Model Editor, Code Generator and Semantic Validator	[47], [36], [42], [23, 24]
Not Presented	[44]

7.1 Tooling of the MDGD Approaches

The tool support of the reviewed MDGD approaches varies from simple solutions only providing a code generator to complex solutions providing a tool suite including model editor, semantic validator and more. Table 8 shows the tools provided by the reviewed MDGD approaches.

In MDD, the model execution tool (either direct or indirect) is essential. Table 8 shows that all the MDGD approaches with tooling information presented have provided some kinds of model execution tools. Most of the approaches provide a *code generator*. The code generator is arguably the most common way of doing MDD [52]. Code generators are implemented based on model transformation technology, which is about transforming one model to another model that can have different syntax and semantics. Model-to-model transformation is the general form of model transformation, and Model-to-text transformation is a special case [61]. Most of the MDGD approaches only use Model-to-text technologies to generate code from game models, and a few approaches have added one or more Model-to-model transformation steps before the Model-to-text transformation, to reduce the semantic gap between the model and the code. In Reference [61], two Model-to-text transformation methods are discussed, which are the Visitor-based method, and the Template-based method. In Reference [52], the Template-based method is further divided into Templates and Filtering Method, and Templates and Meta-model method. For the Model-to-model transformation, Reference [61] pointed out five categories: Direct-Manipulation, Relational, Graph Transformation-based, Structure-driven, and Hybrid.

To implement the above model transformation approaches, one can build them from scratch or preferably use available technology frameworks. Most of the MDGD approaches have made use of existing frameworks to simplify the implementation of a code generator. However, still a few approaches choose to make their own implementation from scratch. Table 9 presents the transformation methods and frameworks used by the reviewed approaches.

The *interpreter* is another mechanism supporting the model-execution, which shares the same underlying principles as the code generation [52]. Only References [25–27] in the reviewed MDGD approaches has chosen to implement an interpreter, and this will be further discussed in Section 7.3.

The *model editor* is also a central tool in MDD [52]. A specialized model editor can better support a modeling language, thus making the development more effective. The evolution of modern language workbenches has reduced the cost of implementing model editors, resulting in that most of the MDGD approaches have provided a model editor as a part of their tooling solution. However, some of the MDGD approaches have not provided any model editors. Among them, References [19, 20] use UML as the modeling language whose models can be created with general UML tools, so a specific model editor is not necessary. Other approaches mostly use textual languages whose models can be created with general text editors. Note that language-specific support such as keyword highlighting, and syntax checking is lacking.

Before generating the code, the correctness of models must be checked with respect to the meta-model, and this checking is in most cases done by a separate tool to avoid the unnecessary

Table 9. Model Transformation Methods and Frameworks

Transformation Method	Transformation Framework	MDGD Approach
Relational Model-to-Model Transformation	Query/View/Transformation (QVT)	[40]
Hybrid Model-to-Model Transformation	Atlas Transformation Language (ATL)	[17], [21, 22], [36], [40], [75]
Visitor-based Model-to-Text Transformation	Xtend	[32]
	Custom Code Generation Program	[28, 29], [37], [74]
Templates and Metamodel-based Model-to-Text Transformation	Epsilon Generation Language (EPL)	[17]
	MOFScript	[19, 20], [30]
	Acceleo	[21, 22], [40], [72, 73], [75]
	Java Emitter Templates (JET)	[33, 34]
	XPand	[36], [14]
	Xtend	[41], [47]
Templates and Filtering Model-to-Text Transformation	FreeMaker	[18]
	XSL Transformation (XSLT)	[31]
	Custom Code Generation Program	[39]
Not Presented	N/A	[23, 24], [38], [42], [43], [44], [45, 46]

Table 10. Tooling Environment of The MDGD Approaches

Category	Tooling Environment	MDGD Approach
Integrated Language Workbenches	Eclipse Modeling Tools	[17], [21, 22], [32], [33, 34], [47], [36], [40], [41], [44], [14], [72, 73], [75]
	Microsoft Visual Studio Tools	[43], [42], [45, 46]
Individual Language Tools	Freemaker, MOFScript, XSLT, DiaMeta	[18], [19, 20], [30], [31], [44]
Game Engines	jMonkeyEngine	[18]
	Unity	[23, 24], [32], [75]
General Programming Tools	Java	[37], [28, 29]
Not Presented	N/A	[38], [39], [74]

complication of the code generation [52]. Four approaches have provided such a tool called a *semantic validator*. A semantic validator can detect semantic inconsistencies and defects within the models. Considering the difficulty in debugging model-driven projects, the semantic validator is very useful. Modern language workbenches simplify the development of the tool, so it is practical to be implemented as part of the MDGD tooling.

7.2 Tooling Environment

The tooling environment of MDGD approaches is about the software used to create and support the MDGD tools. The choice of tooling environment can significantly impact the cost of developing tools. Table 10 presents the tooling environments of the reviewed MDGD approaches categorized into four groups.

The majority of the MDGD approaches used integrated language workbench [62] as the tooling environment. A language workbench is a software suite integrating various essential components

for developing and supporting DSLs. With a language workbench, a complete tool-chain of a MDGD approach can be created in a rapid and simple process.

The most popular language workbench used by the reviewed MDGD approaches is *Eclipse Modeling Tools (EMF)*. Eclipse is an open-source development platform, which has an active community developing various development tools as plug-ins including both traditional development tools and MDD tools, for example:

- Eclipse Modeling Framework (EMF): It includes eCore, a meta-meta-model for defining DSLs, and a couple of generators that generate model editors and a generic editing framework for editing the models [52]. According to the classification of tooling approaches in Reference [48], EMF-based meta-tools enable an approach with the best maturity that is described as "Integrated meta-modeling and modeling environment". EMF is used by References [33, 34], [47], [36], [40], [72, 73], [75], and [44].
- Xtext: It is a framework for developing textual DSLs on the Eclipse platform. The framework provides a language and a user interface for defining the concrete syntax of DSLs. Once a DSL is defined, the framework generates a parser, a class model for the abstract syntax tree, and a textual model editor integrated into the Eclipse platform. The model editor offers syntax coloring, code completion, static analysis, and more. Xtext was used by References [17], [32], [47], [41], and [14].
- ATL Transformation Language (ATL) [63]: It is a Domain Specific Language and corresponding toolkit built on EMF for creating model transformations, that can semantically turn the model in one language to another. ATL is often used for generating intermediate models to facilitate code generation. It is used by References [17], [21, 22], [36], [75], and [40].
- Acceleo: It is a template-based implementation of the Model-to-Text transformation Language (MTL) standardized by OMG, and it greatly reduces the effort of writing a code generator. Acceleo is used by References [21, 22], [72, 73], [75], and [40].

Apart from the above tools, other popular Eclipse-based technologies used include Xpand used by [36] and [14], Xtend/Xtend2 used by References [32], [47], and [41], and ALPiNA used by [36].

Microsoft Visual Studio Tools is another option of a language workbench, which is a powerful toolkit integrating "frameworks, languages, editors, generators, and wizards that allow users to specify their own modeling languages and tools" [48]. The toolkit provides its own meta-meta model for defining DSLs, which is equivalent to eCore in EMF. One limitation is that Microsoft Visual Studio Tools is only available on the Windows platforms. It is used by Reference [42] to create the tool-suite for a DSL including a model editor, a code generator and a semantic validator. References [41] and [45, 46] also use Microsoft Visual Studio Tools.

Apart from language workbenches, individual language tools may also be used in developing DSLs, especially when the language workbenches do not provide the proper components. *Freemaker* used by Reference [18], *MOFScript* used by References [19, 20] and [30], and *XSLT* used by Reference [31] are all tools for creating code generators. DiaMeta is a tool for creating graphical editors, which takes an EMF model as the meta-model of the DSL, and an *editor specification* that is created with DiaMeta tools as the concrete syntax specification. A graphical DSL editor is then generated. DiaMeta is used by Reference [44].

Some MDGD approaches have chosen game engines as the tooling environment, where a plug-in for the game engine is developed, which supports the DSLs of the approaches. Using a game engine as the tooling environment is a double-edged sword, as it simplifies the development toolchain while sacrificing the advantages of dedicated language tools. Unity is a commercial game engine that features a flexible level editor open for extension, and it is used by References [23, 24], [75],

and [32] as the tooling environment. jMonkeyEngine is an open-source engine that is used by Reference [18].

Last, References [37] and [28, 29] directly use Java to create the DSL tools. Compared to the plain programming approach, we believe that the use of language tools can be more productive and easier to adopt changes.

7.3 Summary of MDGD Tooling

The MDGD approaches have provided different levels of tooling support, from the basic code generator to the versatile tool suite. Successful MDD may require more tools that are lacking in current MDGD approaches, and such tools are summarized in Reference [64], which include *model validation tools, model instances management tools, model mapping tools, model-driven testing tools, dashboard applications, version-control and distributed modeling tools*, and *software process management tools*. This section summarizes the challenges with MDGD tools and tooling environments.

7.3.1 Interoperability of MDGD Tools and Game Engine Tools. The game industry has a long history of using game engines including the tool suites coming with the engines. When MDGD is introduced to game production, its tools must be integrated with the engine tools instead of replacing them. However, most of the existing MDGD approaches are only aware of the run-time engine, while ignoring the engine tool suite. Moreover, in some approaches, tools have been developed to re-implement the functionalities provided by the engine tools. For example, a *world editor* is the core tool in most of the engine tool suites, which is re-implemented in SLGML [42], AcadEx [45, 46], and PacMan DSL [44]. In contrast, some approaches have embraced game engines [18], [23, 24], [32] but did not take full advantage of existing MDD tooling environments, which makes their modeling tools difficult to create and maintain. References [72, 73] present an approach addressing this issue, which integrated the game engine tools and MDGD tools with an architecture named Engine-Cooperative Game Modeling (ECGM).

7.3.2 Model-level Debugger. The model-level debugger is the tool that helps modelers to identify semantic errors and locate them at the model level. The static semantics can be checked with the semantic validators as in References [47], [36], [42], and [23, 24], but the dynamic semantic errors have to be identified through observing the models at run-time. Without the model-level debugger, such errors can only be reported from the generated code and cannot be traced back to the model level. This could bring significant problems to the productivity of MDD in general, since extra time has to be spent on locating known code-level errors at the model level, which can be even more expensive than locating known binary-code level errors at the GPL program level, because the semantic gap between the abstraction levels of the modeling language and the generated code is wider [65]. Moreover, we argue that the model debugger is particularly important for MDGD, because an important purpose (or advantage) of MDGD is to enable game designers and level designers to make as much the game logic as possible without the need for help from programmers. However, if (dynamic semantic) errors can only be reported at the code level, then most of this advantage is nullified, as professional programmers are still needed in the modeling process to fix the bugs.

7.3.3 Cooperation of Model Interpretation and Code Generation. As we have discussed, there are two alternatives for making models executable, which are code generation and interpretation [8]. Interpretation tools allow models to be executed without producing intermediate output. One important advantage of interpretation is that "they provide early direct experience with the system being designed" [65]. There are at least two reasons that make interpretation tools particularly valuable for game development: One is that the game development suffers a workflow problem, as

compiling game code can take hours. The other is the modelers in game development sometimes are game designers or level designers who might not be able to use the programming tools such as compilers and binary-image-making tools. If MDGD tools support the cooperation of interpreter and code generator, then it will improve the usefulness of the MDGD approach.

8 EVALUATION METHODS

Most of the MDGD research follows the Design Science Research (DSR) paradigm, which is about "a designer answers questions relevant to human problems via the creation of innovative artifacts" [66]. Such artifacts may include constructs, models, methods, instantiations and better design theories [66]. In short, this definition includes any designed object with an embedded solution to an understood research problem [67]. Evaluation is a central and essential activity in conducting rigorous DSR [68]. All the reviewed MDGD approaches include an evaluation. This section discusses two major aspects of the MDGD evaluation: the purpose and the method.

The purpose of the evaluation is about what quality attributes of the MDGD approach are concerned in the evaluation. The major purposes we found include:

- *Utility:* Whether the MDGD approach is applicable in developing the target games, i.e., whether the approach "works."
- *Productivity:* Whether the MDGD approach can improve the productivity of game development, compared to the traditional development technologies.
- *Usability:* Whether the MDGD approach is easy to use, for example, easy to learn, or can lower the technical threshold of development.

The method of the evaluation is about the approach for doing the evaluation. In Reference [69], 148 publications about DSR in Computer Science and Information Systems were reviewed, and their evaluation methods were categorized as eight types: *Logical Argument, Expert Evaluation, Technical Experiment, Subject-based Experiment, Action Research, Prototype, Case Study,* and *Illustrative Scenario*. This taxonomy framework has covered all the evaluation methods that are used by the MDGD approaches reviewed, so that we will adopt the framework. For the selection of the evaluation methods, Reference [68] provided a comprehensive framework on how to make the decision, and Reference [69] argued that artifact type, context, and data availability are significant factors that affect the selection. For the reviewed MDGD approaches, it is revealed that the data availability may be most important: It is difficult to obtain data from real game projects in the game industry; therefore, most of the researchers have to evaluation their approaches by developing prototypes in research environments. The purpose of the evaluation also affects the evaluation method selection significantly. For example, if productivity improvements are to be evaluated, then an experiment method is often applied; and if usability is the focus, then a case study is usually used.

8.1 Prototype

The Prototype evaluation method is defined in Reference [69] as "Implementation of an artifact aimed at demonstrating the utility or suitability of an artifact." Here, we need to distinguish the game prototype and the technology prototype. The latter is an implementation of the technology that *supports* an MDGD approach, while the former is an illustrative artifact that is produced *using* an MDGD approach. When we discuss the *Prototype* method in this section, we are addressing the technology prototype, while the game prototype is discussed in Section 8.2.

All the reviewed MDGD approaches have implemented some technology prototypes, although they vary a lot in maturity and complexity. A working technical prototype is, first, a

proof-of-concept validation and, second, the base for further evaluation. In all the reviewed MDGD approaches, the *Prototype* method is used in combination with one or more other methods.

8.2 Illustrative Scenario

According to References [69], *Illustrative Scenario* means applying an artifact in a synthetic or real-world situation to demonstrate its utility. An *Illustrative Scenario* is used frequently among the reviewed MDGD approaches, where 16 out of 26 approaches more or less have used the method. In the simplest case, Reference [40] only has described how the approach works with the general maze game concept, without a working game prototype presented, while other approaches, e.g., References [17], [38], and [47] have created one or more game prototypes to demonstrate their capabilities. Furthermore, Reference [39] has developed a couple of commercial mobile games to show its utility in a real-world situation. Apart from the above-mentioned technical approaches, References [14] and [45, 46] focus on the MDGD process, and both demonstrate how the processes work when developing prototypes.

8.3 Subject-based Experiment

Subject-based Experiment is a kind of experiment that involves subjects to evaluate whether an assertion is true or not [69]. In contrast to *the Technical Experiment* method, *the Subject-based Experiment* concerns not only the technology itself but also its effect on the people utilizing the technology. Five out of 26 MDGD approaches used Subject-based Experiment, which are References [18], [19, 20], [21, 22], [72, 73], and [43]. The method is mainly used for quantifying the productivity improvement from applying MDGD. Besides, it can validate the utility of the approaches, since the approaches must be able to function before being tested in experiments. The typical setting of such experiments is to develop the same prototype game using the MDGD and the traditional coding approach, respectively, and then comparing the time used.

8.4 Case Study

The *Case Study* method applies an artifact from DSR to a real-world situation and evaluates its effect on the real-world situation [69]. The real-world situation in our context is mainly the MDGD users, their knowledge, skills, and attitudes. We need to distinguish between a Case Study and an Illustrative Scenario where the former concerns about the effect of the approach on the user and environment, and the latter concerns about how the approach itself works.

A significant advantage of the *Case Study* method is that the results are from real-world application and real users. The method was used by 6 out of 26 MDGD approaches, and the primary purpose of these evaluations was the usability. For example, in Reference [21, 22], three domain experts and one programmer were hired to create games using the approach, and their activities and interactions in development were observed and analyzed, to evaluate whether the approach can fulfill the technical requirements, and how easy it is for non-programmers to use the approach.

In addition to usability, Reference [44] also reported productivity improvements, because in the case study student groups implemented prototypes with the MDGD approach from scratch in hours, although there was no comparable data with coding approach presented. Use of the Case Study method can also validate the utility of the MDGD approaches, because the approaches must work in the first place before affecting the user and environment.

8.5 Summary of Evaluation Methods

The evaluation methods used by the MDGD approaches are summarized in Table 11, with a brief description for each MDGD approach about how the method is used and for what purposes. Note that some MDGD approaches use more than one evaluation method. For example, Prototype is

Table 11. Evaluation Methods Used by The MDGD Approaches

Evaluation Method	Use of The Method	Purpose	MDGD Approach
Illustrative Scenario	Developed a set of tower defense games such as Space Attack to demonstrate the approach	Utility Productivity	[17]
	Used the approach to make a serious game named Friend ATTAC	Utility Productivity Usability	[25–27]
	Used ScriptEase to create game scripts that can replace the hand-written code in NWN modules and calculated the code lines generated.	Utility Productivity	[28, 29]
	Prototyped a maze game with the approach, but no details of the game provided.	Utility	[30]
	Briefly demonstrated how to model the User Interface of an Adventure game	Utility	[31]
	Demonstrated how to add Achievement system to a serious game	Utility	[32]
	Showed the model of a Car Race game	Utility	[33, 34]
	Developed a pervasive game, and calculated the time used on development as well as code lines	Utility Productivity	[47]
	Created a game scenario model and generated JavaScript code	Utility Productivity	[37]
	Developed an educational game for learning fire evacuation	Utility	[38]
	Developed a couple of mobile games and commercially released	Utility Usability	[39]
	Described a generic maze game concept to demonstrate how the approach works	Utility	[40]
	Prototyped a game with Adventure DSL to demonstrate the proposed MDGD process	Utility	[14]
	Created ArcadEx and used it to prototype games to show how the MDGD process works	Utility	[45, 46]
	Rewrote a Location-based Game (AudioRio) with the proposed approach	Utility	[74]
	Implemented a medical training game with the proposed approach to demonstrate how it works	Utility	[75]
Subject-based Experiment	Two student groups (six students for each group) finished game development tasks with traditional technology and the MDGD approach, respectively. The time spent and correctness of answers were compared	Productivity Utility	[18]
	Implemented Bubble Bobble with the MDGD approach and traditional technology, respectively, and compared time used, and calculated the code lines generated.	Utility Productivity	[19, 20]
	A serious game prototype was developed by three domain experts and a programmer with a coding and MDGD approach respectively, and compared the time used	Utility Productivity,	[21, 22]
	(1) Developed Pong game with the coding approach and the MDGD approach, respectively, by the author, and compared code lines and time used; (2) Developed SpaceKartz with the approach by the author, compare with SIG Games on code lines and time	Utility Productivity	[43]
	A game prototype *Orc's Gold* was developed with both traditional approach and the MDGD approach, and the time used as well as code lines were compared.	Utility Productivity	[72, 73]

(Continued)

Table 11. Continued

Evaluation Method	Use of The Method	Purpose	MDGD Approach
Case Study	Observed the development process of making a serious game with the MDGD approach by three domain experts and a programmer, and the behaviors of the participants	Utility Usability	[21, 22]
	(1) Five educators without game development knowledge participated in an experiment to create a serious game; (2) A serious game was developed with the guidance by domain experts and instructors and evaluated by SHaC apprentices.	Utility Usability	[23, 24]
	The approach was introduced to a group of participants to try out	Utility	[41]
	used some student groups to develop three game prototypes	Utility Productivity Usability	[44]
	Developed a game named Ultimate Berzerk with the approach, and observed the process as well as the result	Utility Usability	[42]
	The approach was introduced to five students to try out	Utility Usability	[75]
Prototype	All the MDGD approaches implemented a technology prototype, which provides the proof-of-concept validation and the base for other methods.	Utility	All the MDGD Approaches

the fundamental evaluation method used by all the MDGD approaches, which validates if the approaches can be implemented, providing the base for utility. Illustrative Scenario is mostly used for validating the utility of the approaches. The Subject-based experiment is suitable for evaluating the productivity improvements, while Case Study is the standard method for evaluating usability.

The *Illustrative Scenario* method as the second most popular method, has been used by 16 approaches, but only Reference [39] has been validated in a commercial game development project. The rest of the approaches have only been demonstrated by developing research prototypes. This is because most research projects cannot afford the considerable cost of developing full-scale commercial games. It is also difficult to persuade a game company to apply a research-in-progress approach in a real-world development process because of the high-investment and high-risk nature of the game industry.

Note that the *Illustrative Scenario* method is descriptive and qualitative and cannot produce quantitative results. Due to this limitation, most of the approaches only use the method for validating the utility. Some approaches such as Reference [17] do use the method to evaluate also the productivity or usability, by collecting some data from prototyping (e.g., code lines generated), but the results are not as convincing as quantitative methods like *Experiments*. The *Experiment* method can generate quantitative and relatively generalizable results. However, we notice two significant limitations with the design of the experiments that have threatened the generalizability of the evaluation:

- Small sample size: Only Reference [18] has 12 participants finishing the experiment, while other approaches are evaluated by only one participant or one participant group.
- Loose variable control: All the approaches except Reference [18] use the same participant group to develop the same game prototypes twice with the MDGD approach and coding approach, respectively. The confidence of the results is therefore reduced, since it is likely that the experience from the first-time development makes the development second-time easier.

9 CONCLUSION

Model-driven Game Development (MDGD) brings the general Model-driven Development methodology to the game development domain, which can potentially make game development faster and easier. Apart from the advantages of using MDD in general software development, MDD is particularly useful for game development, because a significant proportion of game developers are non-programmers, meaning that the use of models instead of code will not only make their job more accessible but also save cost in communication. However, as an emerging research field, MDGD still has a long way to go in commercial game development. We have reviewed 26 approaches in this article, but only one of them [39] has been used in commercial games development.

The reviewed MDGD approaches aim at different game domains: The broad domains cover a wide range of games such as *all* 2D games, while the narrow ones only cover one specific game. The domains are also defined using different methods. A domain framework is a software that implements the domain semantics, thus makes the models executable. The domain frameworks chosen by the MDGD approaches vary. In the simple cases, graphics SDKs or OS are used as the domain framework, and in more cases, game engines or modified game engines are used. Model interpreters can also be used as domain frameworks, but only one of the approaches creates an interpreter for the purpose. The modeling languages invented or adopted by the approaches have various forms such as textual, graphical, tree view, and form view. The semantics of the languages cover both behavioral and structural aspects of games. The abstract syntax of the languages also varies, which support different forms of models such as state machines, directional graphs, and decision trees. Tooling is essential for MDD to become useful. There are four kinds of tools supported by the MDGD approaches: Code Generators, Model Editors, Semantic Validators, and Model Interpreters. Most of the MDGD approaches have chosen the Eclipse-based modeling tools as the tooling framework, while Microsoft DSL tools and other tools such as DiaMeta have also been used. Last, evaluation is an essential part of any research. We analyzed the purposes and methods of evaluation of the MDGD approaches and found that there are mainly four methods being used: Prototype, Illustrative Scenarios, Subject-based Experiment, and Case Study. The methods were mainly used for three evaluation purposes: Utility, Usability and Productivity Improvements.

There are still challenges and gaps in the reviewed MDGD approaches. For example, the actual domains supported are usually narrower than the domains claimed to be supported, essential tools such as model-level debuggers are missing, and evaluation of MDGD approaches can also be improved with better processes and designs.

Further work includes exploring more opportunities for MDGD with the inspirations from the literature review in mind and solutions addressing the challenges presented that can potentially improve the usefulness of MDGD in practice.

REFERENCES

[1] J. Blow. 2004. *Game Development: Harder Than You Think*. Queue 1, 10 (2004), 28–37.
[2] D. Callele, E. Neufeld, and K. Schneider. 2005. Requirements engineering and the creative process in the video game industry. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering*.
[3] Richard Rouse III. 2010. *Game Design: Theory and Practice*. Jones & Bartlett Learning.
[4] J. Schell. 2008. *The Art of Game Design A Book of Lenses*. Elsevier.
[5] D. Canere, Eric Neureid and K. Schneider. 2005. Requirements engineering and the creative process in the video game industry. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering*.
[6] Robert France and Bernhard Rumpe. 2007. *Model-driven Development of Complex Software: A Research Roadmap. 2007 Future of Software Engineering*. IEEE Computer Society, 2007.
[7] Walt Scacchi and Kendra M. Cooper. 2015. Research challenges at the intersection of computer games and software engineering. In *Proceedings of the Conference of Foundations of Digital Games*.
[8] M. Fowler and R. Parsons. 2011. *Domain-Specific Languages*. Addison-Wesley.

[9] Kai Chen, Janos Sztipanovits, and Sandeep Neema. 2005. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In *Proceedings of the 5th ACM International Conference on Embedded Software*. ACM.
[10] K. Salen and E. Zimmerman. 2004. *Rules of Play Game Design Fundamentals*. The MIT Press.
[11] A. W. B. Furtado and A. L. M. Santos, 2006. Using domain-specific modeling towards computer games development industrialization. In *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06)*.
[12] A. W. B. Furtado and A. L. M. Santos. 2007. Extending visual studio .NET as a software factory for computer games development in the. In *Proceedings of the 2nd International Conference on Innovative Views of .NET Technologies (IVNET'06)*.
[13] A. W. B. Furtado, A. L. M. Santos, and G. L. Ramalho. 2007. Computer games software factory and edutainment platform for Microsoft .NET. In *IET Software* 1, 6 (2007), 280–293. DOI: 10.1049/iet-sen:20070023
[14] R. Walter and M. Masuch. 2011. How to integrate domain-specific languages into the game development process. In *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology*. ACM, 1–8.
[15] T. Dybå and T. Dingsøyr. 2008. Empirical studies of agile software development: A systematic review. Info. Software Technol. 50 (9–10): 833–859.
[16] A. S. Zahari, L. A. Rahim, and M. Mehat. 2016. A review of modelling languages for adventure educational games. In *Proceedings of the 3rd International Conference on Computer and Information Sciences (ICCOINS'16)*. IEEE, 495–500.
[17] K. Sánchez, K. Garces, and R. Casallas. 2015. *A DSL for Rapid Prototyping of Cross-platform Tower Defense Games*. In *Proceedings of the 10th Colombian Computing Conference (CCC'15)*. IEEE.
[18] E. F. D. Prado and D. Lucredio. 2015. A flexible model-driven game development approach. In *Proceedings of the 9th Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS'15)*. IEEE.
[19] E. M. Reyno et al. 2009. Automatic prototyping in model-driven game development. *Comput.* Entertain. 7, 2 (2009), 1–9.
[20] E. M. Reyno and J. A. C. Cubel. 2008. Model-driven game development: 2D Platform Game prototyping. In *Proceedings of the 9th International Conference on Intelligent Games and Simulation (EUROSIS'08)*.
[21] V. Guana, E. Stroulia, and V. Nguyen. 2015. Building a game engine: A tale of modern model-driven engineering. In *Proceedings of the IEEE/ACM 4th International Workshop on Games and Software Engineering*.
[22] V. Guana and E. Stroulia. 2014. Phydsl: A code-generation environment for 2D physics-based games. In *Proceedings of the IEEE Games, Entertainment, and Media Conference (IEEE GEM'14)*.
[23] N. Thillainathan et al. 2013. Enabling educators to design serious games—A serious game logic and structure modeling language. In *Proceedings of the 8th European Conference on Technology Enhanced Learning (EC-TEL'13)*. Springer-Verlag.
[24] N. Thillainathan and J. M. Leimeister. 2016. Educators as game developers: Model-driven visual programming of serious games. In *Proceedings of the 9th International Conference on Knowledge, Information and Creativity Support Systems (KICSS'14)*. Springer Verlag.
[25] F. Van Broeckhoven and O. De Troyer. ATTAC-L: A modeling language for educational virtual scenarios in the context of preventing cyber bullying. In *Proceedings of the Conference on Serious Games and Applications for Health (SeGAH'13)*. IEEE.
[26] O. Janssens et al. 2014. Educational virtual game scenario generation for serious games. In *Proceedings of the Conference on Serious Games and Applications for Health (SeGAH'14)*. IEEE.
[27] S. Van Hoecke et al. 2015. Enabling control of 3D visuals, scenarios and non-linear gameplay in serious game development through model-driven authoring. In *Proceedings of the 5th International Conference on Serious Games, Interaction, and Simulation (SGAMES'15)*. Springer-Verlag.
[28] M. Cutumisu et al. 2006. Generating ambient behaviors in computer role-playing games. *IEEE Intell.* Syst. 21, 5 (2006) 19–27.
[29] M. McNaughton et al. 2004. ScriptEase: Generative design patterns for computer role-playing games. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*. IEEE Computer Society.
[30] R. E. Vargas et al. 2013. MDA game design for video game development by genre. In *Proceedings of the MODELS-JP 2013 Co-located with the 16th International Conference on Model-driven Engineering Languages and Systems (MODELS'13)*. CEUR-WS.
[31] M. Minovic et al. 2009. Model-driven development of user interfaces for educational games. In *Proceedings of the 2nd Conference on Human System Interactions (HSI'09)*. IEEE Computer Society.
[32] A. Matallaoui, P. Herzig, and R. Zarnekow. 2015. Model-driven serious game development integration of the gamification modeling language GaML with unity. In *Proceedings of the 48th Annual Hawaii International Conference on System Sciences (HICSS'15)*. IEEE Computer Society.
[33] A. Pleuss, D. Gračanin, and X. Zhang. 2011. Model-driven development of interactive and integrated 2D and 3D user interfaces using mml. In *Proceedings of the 16th International Conference on 3D Web technology*. ACM.

[34] A. Pleuss and H. Hussmann. 2011. *Model-driven Development of Interactive Multimedia Applications with MML*. In *Model-driven Development of Advanced User Interfaces*. Springer, Berlin, 199–218.
[35] D. Buchs, S. Hostettler, A. Marechal, and M. Risoldi. 2010. *Alpina: A Symbolic Model Checker*. In *Proceedings of the International Conference on Applications and Theory of Petri Nets*. Springer, Berlin, 287–296.
[36] E. Marques et al. 2012. The RPG DSL: A case study of language engineering using MDD for generating RPG games for mobile phones, In *Proceedings of the Workshop on Domain-specific Modeling*. ACM, 13–18.
[37] M. Stürner and P. Brune. 2016. Virtual worlds on demand? Model-driven development of javascript-based virtual world UI components for mobile apps. In *Proceedings of the 4th International Conference on Model-driven Engineering and Software Development (MODELSWARD'16)*. SciTePress.
[38] E. J. Marchiori et al. 2011. A visual language for the creation of narrative educational games. J. Vis. Lang. Comput. 22, 6(2011), 443–452.
[39] E. R. N. Valdez et al. 2013. Gade4all: Developing multi-platform videogames based on domain specific languages and model-driven engineering. *Int. J. Interact. Multimedia Artific. Intell.* 2, 2 (2013), 33-42.
[40] L. Morales, D. Méndez-Acuna, and W. Montes. 2011. Model-driven game development-case study: An MTC for Maze-game s Prototyping. Revista electrónica en construcción de software 5, 3 (2011), 1–15.
[41] M. Funk and M. Rauterberg. 2012. PULP scription: A DSL for mobile HTML5 game applications. In *Proceedings of the International Conference on Entertainment Computing*. Springer, Berlin.
[42] A. W. Furtado and A. L. Santos. 2006. Using domain-specific modeling towards computer games development industrialization. In *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06)*. Citeseer.
[43] F. E. Hernandez and F. R. Ortega. 2010. Eberos GML2D: A graphical domain-specific language for modeling 2D video games. In *Proceedings of the 10th Workshop on Domain-Specific Modeling*. ACM, 1–1.
[44] S. Maier and D. Volk. 2008. Facilitating language-oriented game development by the help of language workbenches. In *Proceedings of the Conference on Future Play: Research, Play, Share*. ACM, 224–227.
[45] A. W. Furtado et al. 2011. Improving digital game development with software product lines. IEEE Software 28, 5 (2011), 30–37.
[46] A. W. Furtado, A. L. Santos, and G. L. Ramalho. 2011. 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, and VMIL'11*. ACM.
[47] H. Guo et al. 2015. Realcoins: A case study of enhanced model-driven development for pervasive games. Int. J. Multimedia Ubiq. Eng. 10, 5 (2015), 395–411.
[48] S. Kelly and J.-P. Tolvanen. 2008. *Domain-Specific Modeling Enabling Full Code Generation*. John Wiley & Sons.
[49] Hong Guo et al. 2014. PerGO: An ontology towards modelddriven pervasive game development. In *Proceedings of the OTM Confederated International Conferences: On the Move to Meaningful Internet Systems*. Springer, Berlin.
[50] S. Kelly and R. Pohjonen. 2009. Worst practices for domain-specific modeling. IEEE Software 26, 4 (2009), 22–29.
[51] A. W. B. Furtado et al. 2011. Improving digital game development with software product lines. IEEE Software 28, 5 (2011), 30–37.
[52] T. Stahl, M. Voelter, and K. Czarnecki. 2006. *Model-driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.
[53] Holger Krahn, Bernhard Rumpe, and Steven Völkel. 2007. Integrated definition of abstract and concrete syntax for textual languages. In *Proceedings of International Conference on Model-driven Engineering Languages and Systems*. Springer, Berlin.
[54] J. P. Tolvanen and S. Kelly. 2005. Defining domain-specific modeling languages to automate product derivation: Collected experiences. In *Proceedings of the International Conference on Software Product Lines*. Springer, Berlin, 198–209.
[55] M. Petre. 1995. Why looking isn't always seeing: Readership skills and graphical programming. *Commun.* ACM, 38, 6 (1995), 33–44.
[56] L. Engelen and M. v. d. Brand. 2010. Integrating textual and graphical modelling languages. Electr. Notes Theoret. Comput. Sci. 253 (7).
[57] R. Heckel. 2006. Graph transformation in a nutshell. *Electr.* Notes Theoret. Comput. Sci. 148, 1 (2006), 187–198.
[58] S. R. Safavian and D. Landgrebe. 1991. A survey of decision tree classifier methodology. *IEEE Trans.* Syst. Man Cybernet. 21, 3 (1991), 660–674.
[59] D. Fisher. 1966. Data, documentation, and decision tables. *Commun.* ACM, 9, 1 (1966), 26–31.
[60] S. Kent. 2002. Model-driven engineering. In *Proceedings of the International Conference on Integrated Formal Methods (IFM'02)*.
[61] K. Czarnecki. 2003. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model-driven Architecture*.
[62] Sebastian Erdweg et al. The state of the art in language workbenches. In *Proceedings of the International Conference on Software Language Engineering*. Springer, Cham.

[63] F. Jouault et al. 2006. ATL: A QVT-like transformation language. In *Proceedings of the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*. ACM.
[64] S. Kent. 2002. Model-driven engineering. In *Proceedings of the International Conference on Integrated Formal Methods*. Springer.
[65] B. Selic. 2003. The pragmatics of model-driven development. IEEE Software 20, 5 (2003), 19–25.
[66] A. Hevner and S. Chatterjee. 2010. Introduction to design science research. In *Design Research in Information Systems: Theory and Practice*. Springer, 1–8.
[67] K. Peffers et al. 2007. A design science research methodology for information systems research. J. Manage. Info. Syst 24, 3 (2007), 45–77.
[68] John Venable, Jan Pries-Heje, and Richard Baskerville. 2012. A comprehensive framework for evaluation in design science research. In *Proceedings of the International Conference on Design Science Research in Information Systems*. Springer, Berlin.
[69] K. Peffers, M. Rothenberger, T. Tuunanen, and R. Vaezi. 2012. *Design Science Research Evaluation*. In *International Conference on Design Science Research in Information Systems*. Springer, Berlin, 398–410.
[70] S. Tang and M. Hanneghan (2011). State-of-the-art model-driven game development: A survey of technological solutions for game-based learning. J. Interact. Learn. Res. 22, 4 (2011), 551–605.
[71] B. Roungas and F. Dalpiaz (2016). A model-driven framework for educational game design. In *Proceedings of the 4th International Conference on Games and Learning Alliance*. Springer-Verlag, 1–11.
[72] M. Zhu et al. 2016. Engine- cooperative game modeling (ECGM): Bridge model-driven game development and game engine tool chains. In *Proceedings of the 13th International Conference on Advances in Computer Entertainment Technology*. ACM, 1–10.
[73] Meng Zhu and Alf Inge Wang. RAIL: A domain-specific language for generating NPC behaviors in action/adventure game. In *Proceedings of the International Conference on Advances in Computer Entertainment*. Springer, Cham.
[74] C. Ferreira et al. 2017. A model-based approach for designing location-based games. In *Proceedings of the 16th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames'17)*.
[75] N. Aouadi et al. 2016. Models and mechanisms for implementing playful scenarios. In *Proceedings of the IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA'16)*.
[76] C. Nolêto et al. 2015. An authoring tool for location-based mobile games with augmented reality features. In *Proceedings of the 14th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames'15)*.
[77] Gabor Karsai et al. 2014. Design guidelines for domain specific languages. Arxiv Preprint Arxiv:1409. 2378.
[78] Tony Clark, Andy Evans, and Stuart Kent. 2002. Engineering modelling languages: A precise meta-modelling approach. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*. Springer, Berlin.

Received September 2017; revised August 2019; accepted August 2019