Noname manuscript No.
(will be inserted by the editor)

A Model-Driven Framework for Developing Android-based Classic Multiplayer 2D Board Games

Mohammad Derakhshandi · Shekoufeh Kolahdouz-Rahimi · Javier Troya · Kevin Lano

Received: date / Accepted: date

Abstract Mobile applications and game development are attractive fields in software engineering. Despite the advancement of programming languages and integrated development environments, there have always been many challenges for software and mobile game developers. Model-Driven Engineering (MDE) is a software engineering methodology that applies software modeling languages for modeling the problem domain. In this paradigm, the code is to be automatically generated from the models by applying different model transformations. Besides, manipulating models instead of code facilitates the discovery and resolution of errors due to the high level of abstraction. This study presents an approach and framework, called *MAndroid*, that generates Android-based classic multiplayer 2D board games in a fully automated fashion, relying on the concepts of MDE. Structural and behavioral dimensions of the game are first modeled in MAndroid. Models are then automatically transformed to code that can be run on any mobile phone and tablet running Android 4.4 or higher. In order to evaluate the proposed approach, three board games are fully implemented. Additionally, applicability, developer performance, simplicity and attractiveness of MAndroid are evaluated through a set of questionnaires. MAndroid is also evaluated technically by comparing

M. Derakhshandi and Shekoufeh Kolahdouz Rahimi
MDSE Research Group,
Dept. of Software Engineering,
University of Isfahan, Isfahan, Iran
E-mail: m.derakhshandi71@gmail.com, E-mail: sh.rahimi@eng.ui.ac.ir

Javier Troya
SCORE Lab, I3US Institute
Universidad de Sevilla, Seville, Spain
E-mail: jtroya@us.es

Kevin Lano
Department of Informatics,
King's College London, UK
E-mail: kevin.lano@kcl.ac.uk

it to other Android game-development frameworks. Results demonstrate the benefits of using MAndroid.

Keywords Model Driven · Android · 2D Games · Board Games · Multiplayer Games.

1 Introduction

The mobile software development industry has been experiencing a significant growth since the increasing demand in using smartphones [1]. Due to the existence of different platforms and devices, the industry has faced a variety of challenges, such as time constraints in releasing smartphones applications in the market [2]. The mobile game industry is one of the most popular sectors in software industry, which typically requires large teams composed of members with different specializations for developing sophisticated games [3]. Although game engines have been able to reduce some of their complexities, game development continues to be a complex and time-consuming process that requires many resources [4].

Model-Driven Engineering (MDE) is a methodology targeting at dealing with the system at a high level of abstraction by using models and domain-specific modeling languages and automatically generating code by applying different model transformations [5]. Models can be reused across similar problems by only applying slight changes [5]. Applying model-driven techniques reduces the complexity in game development by setting the focus at a higher level of abstraction than when programming the game [6].

Android is an open-source operating system for mobile phones, developed by Google [7], which has become the most popular mobile platform nowadays [8]. In this paper, a comprehensive approach for automatic generation of Android-based classic multiplayer 2D board games is presented. The domain of board games is widespread nowadays, and it is not trivial to set its boundaries. In this research we focus on classic board games, understood as those that exist for a long time and are played on wooden (or similar materials) boards with beads and dice [9]. Additionally, we focus on multiplayer games that are run on a single mobile phone—a classification of multiplayer games is provided in the background section. In this type of games, two-dimensional graphics are used to represent images and animations. Narrowing the type of game allows to generate 100% correct code by applying an MDE methodology. Our approach is supported by a framework called *MAndroid*. The approach is divided into two general stages. The first stage deals with the modeling of the menu and user interface (UI) of the game, while in the second stage the structure and behavior of the game are modeled. The game menu is a set of graphic elements that build the UI without considering the type and style of game. It is created from a set of pages and every page contains a set of graphic elements. Our approach proposes a novel method for designing these pages. Arrangement possibilities of graphic elements provide a convenient way for designing the UI by considering the concepts related to UI design in Android, such as layouts. The structural properties of the games have been defined in a so-called structural metamodel, by considering different elements such as board, tile, dice and player. This is followed by establishing the behavioral properties, in which the rules and methods for the interaction between players and the structural properties of the game are defined.

The metamodeling language Ecore has been used to build the metamodels in our approach. Ecore is integrated in the Eclipse Modeling Framework, which makes it possible to use any Eclipse-based tool [10]. For instance, the Acceleo model-to-text transformation language has been used for the automatic transformation of Ecore models to Android code [11]. As a final step of the process, the generated code can be compiled in an Integrated Development Environment such as Android Studio to build an executable Android application.

The MAndroid approach and framework are evaluated from different perspectives. First, three classic multiplayer 2D board games, namely *Backgammon*, *Othello* and *Tic-Tac-Toe*, have been fully implemented using the framework. Also, a practical workshop was conducted, in which Tic-Tac-Toe was developed by a number of graduate and post-graduate students using our MAndroid platform. Some questionnaire forms were designed so that students and some Android experienced programmers could evaluate the advantages and disadvantages of the provided framework in comparison to other game development frameworks. The Spearman's correlation coefficient is applied to measure the linear relationship between variables. This indicates the positive or negative effect of two variables on each other. Results of the evaluation show the effectiveness of MAndroid in developing Android-based classic multiplayer 2D board games.

The remaining of this paper is organized as follows. Section 2 sets the context of this research by presenting its background. Section 3 compares our proposal with several related works. Then, Section 4 explains the proposed MAndroid approach and framework. The evaluation of MAndroid is explained in Section 5, while Section 6 presents its results. Finally, Section 7 concludes the paper and describes future works.

2 Background

This section provides a general overview and sets the context of our approach. First, a general description of video games is provided, followed by a summary of Android concepts. The section ends with an introduction to MDE and its main concepts.

2.1 Video Games

A game is a physical or mental competition, which is based on the sets of rules for achieving a specific goal [12]. Video games[1] are applications developed for personal entertainment that are based on the interaction of one or more persons with electronic devices such as personal computers, mobiles or game consoles. In most cases, computer games provide virtual environments, which enable players to control one or more characters based on the game goal [2]. Due to the existence of different platforms, programming languages and technologies, developing video games is a complicated task [13]. For game development, it is essential to count on people with different background and expertise in the teams, including designers of game story, graphic designers, stage designers and, of course, programmers. It would be a challenging task to develop a game if there was not a consistent and close collaboration between team members [14]. Therefore, to enable convenient ways of game development, different tools like game creation tools and game engines have been created and put available in the market. These tools enable to design games for different platforms. Some popular engines are *GameSalad*, *GameMaker*, *Unity* and *Unreal* [15].

Despite these game tools provide some facilities to users, they also have some limitations. In most cases it is not a trivial task to learn to work with such engines, specially for elementary users [15]. Additionally, the code generated by such engines is very generic and the user needs to change some parts of the code to make it applicable for a particular game and platform [16]. The main focus of these engines is on characters and game stages, while they do not concentrate on other parts of the game such as the menu. Therefore, in most cases the games follow the same structure in terms of menu and appearance if they are developed with such engines [15].

2.1.1 Classification of Video Games

In most related works, video games are classified according to their style and graphics. However, in this paper, apart from the style and graphics, the number of players is also considered. In the following, three different dimensions for video games are described:

- **Classification based on game graphics**. In this classification games are categorized into two dimensions (2D) and three dimensions (3D). 2D games have length and width as physical dimensions, which lie on the flat surface. 3D games, in turn, have weight or height as a third dimension. Applying an additional dimension to present elements in 3D games provides more realistic and natural presentation of the elements [17].
- **Classification based on game style**. In this category, games are divided into different parts based on their goal, the type of interaction of players

[1] Throughout this paper, we refer to *video games* also as *computer games* or simply *games*

with the game and the angle of player vision. Some of the most popular game styles for players are Role Playing, Shooter, Board, Simulation, Adventure, Educational based, Platform and Strategic [17,4].
- **Classification based on the number of players**. In this category, games are classified according to the number of players that can compete or collaborate in the game at the same time. Games are classified as one-player or multiplayer. Multiplayer games are typically divided in two types. While some games are run on a single device, some others are run on a network [17,4].

In this paper we concentrate on classic multiplayer 2D board games that are run on the same device with a shared screen page. By limiting the research to one category of classic board game, it will be possible to generate 100% correct code by using an MDE approach. The investigation of other multiplayer games is left as future work.

2.1.2 Board Games

A classic board game or a table game is a type of game played on a board. This kind of games has been designed since the ancient time. With the advent of computers and computing platforms, board games started to have their virtual version as computer games, and they started to get popular among different types of players. With the advances in computing technologies and the creation of new platforms, these virtual board computer games implementations spread across different platforms, reaching inevitably mobile platforms as well [9].

In this type of virtual games, pieces are placed on a board based on a predefined set of rules. Game rules define the goals that players must achieve, such as defeating the opponents, wining a physical position or earning points. Although simple rules are commonly defined for this type of games, users require thoughtful strategies to win the game. Board and pieces are the main elements of any classic board game. Additionally, dice and cards are typically used to provide games with randomness and are important. Also, players in board games get to take their turn [9].

Three different categories have been defined for classic board games. War type is the first category, in which the main goal is to reach the opponent or destroy the opponent's pieces. Popular games in this category are *Chess* and *Go*. In the second category, named race games, the goal is to reach a specific position on the board, such as in *Backgammon* and *Marpele* games. In the third category, so-called alignment category, players focus on generating a particular pattern on a board. *Tic-Tac-Toe* is one of the most-played games in this category [9]. Our approach considers these three categories of classic board games.

Please note there are many more board game categories other than the ones considered in this approach. For instance, we do not consider games such as *Monopoly*, where cards are used, or games like *Carcassonne*, where the board construction is an integrated part of the game. In this paper we only focus on classic board games.

2.2 Android Overview

Android is an open-software stack system based on a Linux Kernel, which is primarily designed for portable devices like smartphones and tablets. Nowadays, it is getting more popular and is being applied in different devices such as smart TVs, automobiles and smart watches. Android is an operating system very popular in the market, with 29 different versions as of 2015 [18]. This paper is focused on the automatic generation of code for classic multiplayer 2D board games in the Android platform.

The Android architecture includes the Linux kernel, Hardware Abstraction Layer, Android Runtime, Native C/C++ Libraries, Java API Framework and the applications layer, which is on top of all those layers. The layers are integrated to provide application development and execution environment on mobile devices [18,19]. There are five fundamental elements for development of Android applications [19]:

– Management of Activity and Fragment. These two elements are used for developing application user interfaces. The application life cycle is related to the life cycle of Activity and Fragment.
– View. This element is used for creation of user interface component in the page.
– Notification Management. This element manages notifications, which are sent from the application to the users.
– Content provider. It enables different applications to share their data.
– Resource management. This element enables to apply other resources to code, such as graphics and strings.
– Intent. It presents a mechanism to transform data between applications and components.

2.3 Model-Driven Engineering

Model-driven engineering (MDE) is a revolutionary paradigm for development of software engineering artifacts. The main artifact in this methodology is the model. A model is a representation of the system at a right level of abstraction. Using models and focusing on a higher level of abstraction than the code provides a better understanding in the stages of design and implementation [5]. Additionally, the number of faults, development time and costs are decreased using this paradigm [15].

Figure 1 presents the common MDE methodology in which the Computational Independent Model (CIM) of the system is firstly defined. This model is designed by domain experts and modelers after analyzing the system requirements. Following this stage, the Platform Independent Model (PIM) with more details is generated by applying corresponding model-to-model transformations. This model requires to be refined for the later generation of the appropriate code. With this aim, by applying different transformations to this model, the Platform Specific Model (PSM) is generated, which contains details

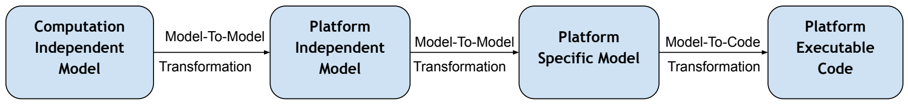

Computation Independent Model
Model-To-Model Transformation
Platform Independent Model
Model-To-Model Transformation
Platform Specific Model
Model-To-Code Transformation
Platform Executable Code

Fig. 1: Architecture of MDE [5]

of the implementation platform. Finally, it is possible to generate code from this model by applying different model-to-text transformations [15].

2.3.1 DSLs vs GPLs in MDE

It is possible to model a system using general purpose languages (GPL) like UML or domain-specific languages (DSL), which are targeted at specific domains.

DSLs are closer to the problem domain and designed for particular purposes at a high level of abstraction. With the advent of MDE technologies, DSLs are developed in industrial and academic organizations for tackling specific issues [20]. For instance, in [21] a DSL for automatic generation of Android location-based applications is developed. It is less productive for a novice user to use a GPL like UML for modeling Android location-based concepts, since they need to spend more time to get domain-specific knowledge. DSLs have three main parts, namely abstract syntax, concrete syntax and semantics. The abstract syntax identifies the language concepts and the relation between them, they are expressed through metamodels. The concrete syntax defines sets of rules for presentation of the abstract syntax. Finally, semantics provide an understanding and appropriate meaning of the language [22], and a way to express it is by means of model transformations.

2.3.2 Metamodels in MDE

As in programming languages each program follows the grammar of the language, in MDE a model needs to follow the rules and constrains of a metamodel, which defines the abstract syntax of the language. A model must conform to a metamodel, which has higher abstraction level than the model [15]. Eclipse Modeling Framework (EMF) is a commonly used modeling framework for definition of metamodels [23], and it is the one used in our approach.

2.3.3 Model Transformations in MDE

A model transformation is a mapping from an input artifact to an output artifact considering sets of predefined rules. Four types of transformation are defined in MDE, which include model-to-model, model-to-text, text-to-model and text-to-text transformations [24]. Transformations play a cornerstone role in MDE, since the correctness of the software artifacts produced by model transformations depends on the correctness of model transformations themselves.

In a model-to-model transformation process, one or more input models, which conform to one or more input metamodels, are transformed to one or more target models that conform to one or more target metamodels. Transformation rules are executed by a transformation engine and need to conform to the model transformation language [25]. Regarding model-to-text transformations, they take one or more models as input and generate one or more text artefacts. They are typically based on the use of templates [26].

2.3.4 Acceleo Framework

Acceleo is an open-source technology and a practical implementation of the *MOFM2T* model-to-text transformation language. This technology provides a pattern-based language for defining patterns and code generation environments. Acceleo has a text editor, which is added as an Eclipse plugin to enable functionality for instruction highlighting, error discovering, content assist and code completion, among others. Profiler is an alternative feature in Acceleo, which presents the number and time of executed instructions within a transformation [27]. Acceleo is the technology used in our approach to define and execute model-to-text tranformations.

3 Related Work

Before presenting our approach, let us present some related works. We divide them in two blocks. The first one includes works related to the application of MDE for development of Android applications, while the second block describes works that apply MDE for the development of 2D games.

3.1 Applying MDE for development of Android applications

After the release of the first official Android operating system, several papers have been published for applying MDE in the design and implementation of Android applications. In [28], 30 studies that applied MDE for development of mobile application are investigated. Among these studies, 11 MDE techniques, 21 tools and 8 modeling languages are identified. The authors classified the selected studies based on different criteria, including mobile platform supporting, modelling and transformation technologies. The paper identifies the potential application of MDE for automatic generation of Android-based applications in different domains. Usman et al. [29] introduced a method for automatic development of cross-platform mobile phone applications. In this paper, each application is modeled both statically and dynamically. In the static vision, the structural features of the system are emphasised using class diagrams, while in the dynamic vision, dynamic behaviors of the system by using state machines are modeled. Heitkotter et al. [30] proposed the MD^2 approach, which provides an automatic generation of Android applications for different platforms. In this approach, the application is modeled using a DSL, and then the code is generated automatically for the corresponding target platform. In [31] an automatic approach is proposed for modeling the application's structure and behavior using class and sequence diagrams, respectively. GenCode [32] is applied for the generation of Android code.

The Applang DSL for description of features in Android applications is presented in [33]. Additionally, Ko et al. [34] extended the UML standard to support modeling features of Android applications. However, code generation is not the main focus in these two papers. Automatic generation of user interfaces for Android applications by focusing on object diagrams using platform-independent models (PIM) is presented in [35]. The PIM is transformed to the platform-specific model (PSM) and then, by using different model-to-text transformations, the graphical user interface of the Android application is generated automatically. A DSL for automatic generation of GUI independent of the target platform is introduced in [36]. However, the approach is limited to support GUI features and currently supports Android platform. Lachgar and Abdali [37] introduced a new approach for modeling and automatically generating sensor-based mobile phone applications. A metamodel specific to this domain is developed. In [21] a model-driven framework for automatic generation of location-based application is developed. It allows novice users to develop an application and customize it according to their specific requirements. The framework is equipped with a DSML, graphical editor and eclipse plugin for generation of final application from models. The generalizability of the framework is presented by developing four industrial location-based applications. Additionally, the usability of the approach is proved through three experiments.

In [38], the authors propose a model-driven framework of MAML in a platform-agnostic fashion with the aim of simplifying the technical complexity by introducing a graphical-domain specific language. This framework enables automatic generation of code for different platforms including Android. The provided functionalities of this framework are limited and further extensions are required for it to become more applicable. A modeling language is introduced in [39] to enable automatic generation of native Android and iOS applications. In this approach it is possible to model the layout and behavior of applications. However, the proposed metamodels are limited and some features are not supported. In [40] an MDA approach for development of mobile application by focusing on data access design is presented. The concepts of data persistence are considered in this research to achieve an offline application. Transformation rules provide automatic generation of Android and Windows phone applications with respect to data persistence and provider metamodels.

3.2 Applying MDE for development of 2D games

In [2] an MDE approach for the automatic generation of 2D games in different platforms by considering five different styles is proposed. The Gade4all graphical tool is developed to provide a user friendly environment that enables users to implement features and rules of the game without writing any line of code. Additionally, it is possible to edit and change the generated code. However, there is no evaluation in this paper to reflect the effectiveness of the proposed approach. Additionally, it does not consider multiplayer games and the provided metamodel is specific to five particular types of games. Some issues are addressed in [15], where the architectural concepts of MDE are considered deeply and a graphical DSL is developed. More information about the transformations is provided, and the evaluation identifies the positive features of this approach. However, this research is also limited to five specific styles and a single-player game.

Sanchez at al. [16] used MDE and DSL techniques for automatic generation of cross-platform applications for tower defense games. In this approach, the dynamic and structural behavior of the system are separated and implemented using a textual DSL. The structural and behavioural models are merged and the architecture model of the system is generated. The code is then provided using model-to-text transformations. ATL, EGL and Xtext languages are used as model transformation languages. The evaluation is based on the portability and productivity of the approach. However, in this approach a graphical DSL is not provided, which may make it difficult for some users to use it. Supporting a single style of the game is an alternative limitation in this research. Additionally, the metamodel is not complete and it is not possible for the user to add new constraints in the game. Considering test cases with different constraints would enhance the applicability of this approach.

PhyDSL, for designing 2D physics-based games, is developed in [41]. It is possible to generate semi-automatic Android code in this platform. This approach enables the user to create a prototype of the game before delivering the product to the market. However, a graphical user interface is not provided for this approach, which makes it difficult for the users to work with it. Additionally, supporting different event controlling for touching screen pages such as *Long Touch*, *Double Tap* and *Swipe* are not provided. It only covers one style of game and a comprehensive evaluation to highlight the strength of the approach is not provided. This is a semi-automated approach and does not provide enough test cases. Marques et al. [42] proposed an approach for the generation of the RPG games for mobile phones. The game is developed by domain analysis, design and implementation in this approach. The proposed metamodel is based on the concept and logic of the games. Additionally, appropriate model-to-model and model-to-text transformations using ATL and Xpand are provided. However, there is no case study in this research to evaluate the effectiveness of the approach. Furthermore, this approach does not cover multiplayer games. Finally, it is a semi-automatic approach and the complete code is not generated.

Table 1: Comparing MAndroid with other approaches

Paper	DSL	Case study	100% code generation	Android support	Board games support	Multiplayer game support	Evaluation
[2]	✗	✓	✓	✓	✗	✗	✗
[6]	✓	✓	✗	✗	✗	✗	✓
[15]	✓	✓	✓	✓	✗	✗	✓
[16]	✗	✗	✓	✓	✗	✗	✓
[41]	✗	✗	✗	✓	✗	✗	✓
[42]	✓	✗	✗	✓	✗	✗	✗
[43]	✓	✗	✗	✗	✓	✗	✗
[44]	✓	✗	✗	✗	✗	✗	✓
MAndroid	✓	✓	✓	✓	✓	✓	✓

A simple DSL for 2D games is proposed by Hernandez and Ortega [6]. A graphical tool is generated in this approach. It is evaluated using time and LOC features. However, no particular style is selected in this approach, therefore the generated code is not complete and it needs some modifications for completion. Altunbay et al. [43] applied MDE for generating 2D board games. The modeling language and model-to-text transformations are provided in this research. The chess game is implemented and sets of patterns are applied to provide the code automatically. However, the provided metamodel is not specific to the mobile domain. Therefore, the generated code is not complete and it requires some changes to be adapted to a particular domain. The evaluation is a bit incomplete, with only one case study provided.

Reyno and Cubel [44] proposed a new approach for developing 2D games using MDE techniques. In this approach, first the PIM of the game is developed, which abstracts structure and behavior of the game. Then the PIM is transformed to a PSM. Following this, a prototype of the game is created and finally the C++ code is generated semi-automatically. The structure of the game is modeled with class diagrams, while the behavior is modeled with state machines. Two instances of a game are implemented using this approach. However, the provided graphical tool in this approach does not cover the behavior of the game. This approach is not specific to mobile platforms and therefore the code requires modification. Additionally, it only supports single-player games.

In summary, although different approaches have been proposed for applying MDE concepts in the development of 2D games, the proposed metamodels are typically not comprehensive and therefore the generated code is not complete [16] [41] [42] [6] [43] [44]. Additionally, in some cases the approach is not limited particularly to Android [43] [44] [6]. Only in [44] the board game is the main focus of the research, but only from a structural dimension. The provided evaluations in most of the approaches are not comprehensive and do not reflect their effectiveness [16] [41] [42] [2]. None of the aforementioned approaches focus on multiplayer games. Furthermore, the graphical user interface is not provided in [42] [2] [43], which makes it difficult for users to work with the

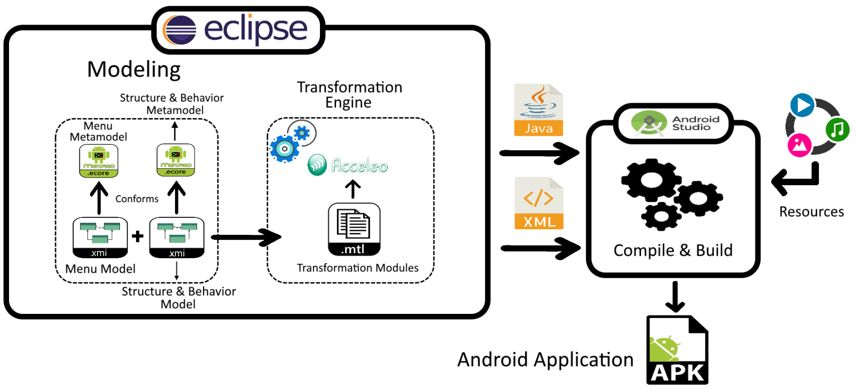


Fig. 2: The Process of game development with MAndroid Approach

approach. Additionally, the behavioral part of the game is not considered in most approaches [44].

Table 1 compares the related works which use MDE for the development of 2D games in terms of whether they propose a DSL, any case study for evaluation, they generate 100% correct code, they support multiplayer games and whether they evaluate the approach. The last row of the table identifies the MAndroid framework, which covers all aspects. As we describe in the following section, in this research we focus on both behavioral and structural dimensions of classic multiplayer 2D board games. The metamodels are not specific to a particular game and cover all the aspects of classic board games. The provided graphical metamodel enables the user to work conveniently with this framework. Additionally, three different board games have been modeled and their code obtained automatically. The generated code is 100% correct and does not require any modification.

4 Approach and Framework

This section presents the model-driven engineering (MDE) methodology and approach for the generation of Android-based classic multiplayer 2D board games. We limited this research to classic board games, which exist from a long time ago and are played on a typically wooden (or made of similar materials) board with beads and dice. Limiting the domain is essential in order to generate functional code automatically [5]. Figure 2 displays an overview of the approach. It can be generally divided in two stages. The first stage applies model-driven techniques, comprising (meta)modeling and transformation tasks, and is carried out on the eclipse modeling framework (EMF)[2]. The second stage involves compilation and building of the final Android code and creation of executable Android APK files, which is carried out in Android Studio.

In the following, the first stage is described in detail. The presentation is divided in three parts. First, the metamodels for user menu, game structure and game behavior are partially shown and explained—a detailed explanation of the metamodels is available in Appendix A. Then, instances of these metamodels are defined. Finally, it is explained how models are transformed to Android executable code by means of model transformations, which are defined and executed in the Acceleo engine[3].

4.1 Metamodels

The definition of games with MAndroid covers the definition of the user interactive menu as well as the game structure and behavior. Two metamodels have been defined for these tasks. First, a metamodel encompasses all necessary concepts for building the game menu. A different metamodel is used to define all concepts and features needed for defining the structure and behavior of the games. Please note both metamodels are independent and not coupled. This means that the game menu is defined without taking into account the game structure and behavior, and the other way around. This simplifies the game modeling process. Besides, user menu and game structure and behavior configurations can be reused across different games. Both metamodels can also evolve independently from each other, since it may be needed to add some extensions in either of them for future versions of Android. Both metamodels are summarized in the following, and a detailed explanation is given in Appendix A.

4.1.1 Game Menu Metamodel

All concepts needed to build a game menu are defined in this metamodel. These concepts and parameters are independent of the nature of the game and can be reused for different Android-based games. Figure 3 displays the menu metamodel, where all attributes have been removed for simplicity. The complete metamodel is shown in Figure 21.

The ***Game*** is the root component of the metamodel. A *Game* contains ***Variable***s, whose purpose is to give the user the ability to change desirable components placed in the game, allowing to use not only predefined settings. It can also contain a ***Resource*** that, in turn, can be composed of different resource types (images, sounds and XML files). All these resources must be later materialized as separate files to be used to build the game menu. In the current

[2] `https://www.eclipse.org/modeling/emf`

[3] `https://wiki.eclipse.org/Acceleo/Specification`

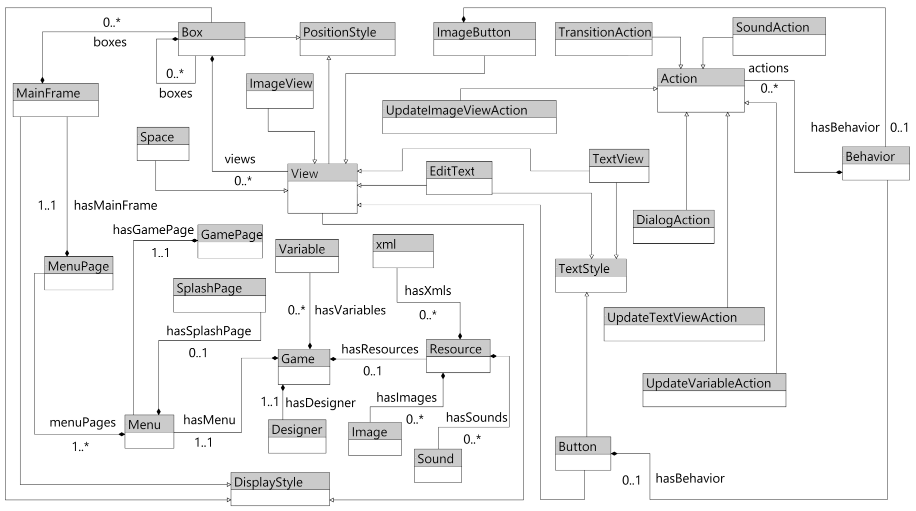


Fig. 3: Excerpt of the Menu Metamodel

version of the approach, the naming of these resources (see *name* attribute in Figure 21) must follow Android OS rules for file naming. As a next step in our implementation, we plan to automatically generate valid names from the name introduced by the user. The ***Designer*** class stores basic information about the game's designer.

A Game also has a ***Menu***, which is composed of a combination of pages. A *Menu* always has one ***GamePage***, a number of ***MenuPage***s, and it can have a ***SplashPage***. The *SplashPage* is an optional page that is displayed at the beginning of the game, offering information such as name and version of the game. *MenuPage*s are like a canvas, where the user can apply different graphical elements. Finally, the *GamePage* is the page where the board game is played. Figure 4 shows all the pages of the Tic-Tac-Toe game developed using MAndroid. As shown in the figure, the game is built using three pages, including, from left to right, a *SplashPage*, a *MenuPage* and a *GamePage.*

Every *MenuPage* has a ***MainFrame***, which acts as a horizontal or vertical frame that holds all other components of the game menu, such as control components. A *MenuPage* can have a number of ***Box***es. We introduce in MAndroid a method called the *"Box Design Method"* to design the user interface of every page. This enables the user to design the user interface of every page without being involved in the technical details related to Android user interface designing such as layouts. Most of the designs that are built using *LinearLayout*, *GridLayout* and *TableLayout* in Android will be implementable using this method. This method divides every page into a set of columns and rows, which are placed together to build a game page. Then, a concept named *Box* is used to place various graphic components in the page. Every box is in fact a space in the page that can hold a graphic component. Adjusting the

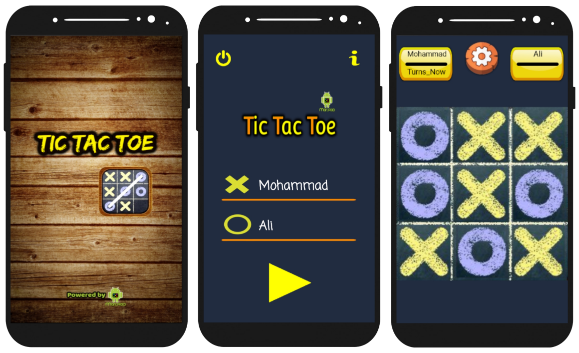


Fig. 4: Pages used in the Tic-Tac-Toe game generated with MAndroid

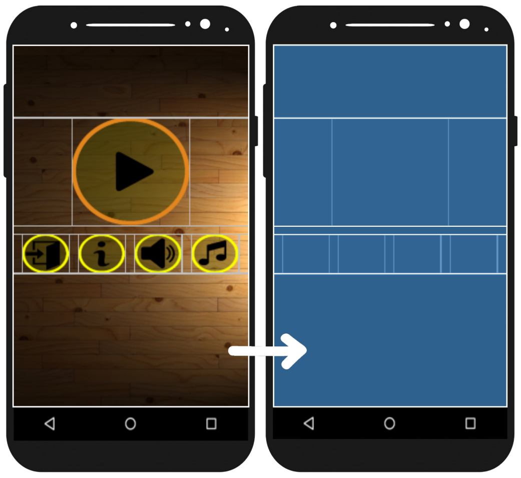


Fig. 5: A design based on *Box*es of a *MenuPage* in MAndroid

size of these boxes on each page is done in proportion to the display size, so that the design can be properly displayed on devices of different sizes and responsive designs can be created. Also, *Box*es can contain other *Box*es to create more complex designs. Figure 5 shows how a *MenuPage* is displayed using this method. The page is made up of 5 different rows and each row holds a number of *Box*es as well. For example, the second row from the top holds three *Box*es. The *Box*es on the right and left are empty spaces that have been placed on both sides to align the row. The *Box* in the middle has been placed as a space to hold a button.

Each *Box* can be composed of ***View***s, which represent the components being displayed in the *Box*. Since the *GameMenu* does not require too many *View* components, we have used and defined only a selected number of Android *Views* in this metamodel, including ***Button*** for creation of buttons,

ImageButton for creation of images as a button, ***ImageView*** for showing images, ***TextView*** for showing text, ***EditText*** for getting input data from the user and ***Space*** to create blank areas inside a box.

In order to present visual features of *View*s, three different classes are abstracted, namely ***DisplayStyle***, ***PositionStyle*** and ***TextStyle***. Each of these is used for specifying properties of the different view components. For instance, *DisplayStyle* allows setting features such as background image or color, *PositionStyle* is used to specify properties related to physical position, and *TextStyle* defines properties like color, font or size of text. Figure 21 displays the attributes of these classes.

Components with which the user is able to interact, such as pressing a button, must have a ***Behavior*** associated. A *Behavior*, in turn, is specified by a set of ***Action***s. As examples of *Action*s, ***UpdateImageViewAction*** is used to change the image of a display component, and ***SoundAction*** allows to perform audio actions, such as reducing or increasing the volume and playing/pausing music, while ***DialogAction*** is used to display an “about” window or a message in the game menu. Regarding updating actions, ***UpdateVariableAction*** is used to update and refresh the value of the variables defined in the game menu, while ***UpdateTextViewAction*** may be used to change the text displayed by a *TextView* component in the game page.

Finally, since this approach is based on a set of pages, it must be possible to shift between pages. This can be a shift between the current page and the next or previous ones, or to a specific page number, to the *GamePage* or exiting the game. This is managed by the ***TransitionAction*** component.

4.1.2 Game Structure and Behavior Metamodel

In the game structure and behavior metamodel, the basic parts of the game that include the behavior and structure of 2D board games are defined. As opposed to the menu metamodel, this metamodel is completely related to the domain of classic multiplayer 2D board games based on Android. Also, in the menu metamodel, the gaming style was not of much importance and it was possible to use the same menu for building other Android games. In contrast, in the structure and behavior metamodel, structural features of 2D board games such as the board display settings and game rules are abstracted. Figure 6 displays the metamodel proposed for the game behavior and structure, where all attributes have been removed for simplicity. The complete metamodel is shown in Figure 22.

Like in the previous metamodel, ***Game*** is the root class, it provides integration between the menu and game metamodels to have a uniform understanding between them. A *Game* has some general features stored in ***GamePlayFeatures***, such as number of players or type of dice.

As mentioned earlier in the game menu metamodel, the declaration of a user interface related to the process of a board game is performed in the *GamePage* page. Because of the differences between the implementation techniques in static and dynamic user interfaces, all the graphic coordinates in the

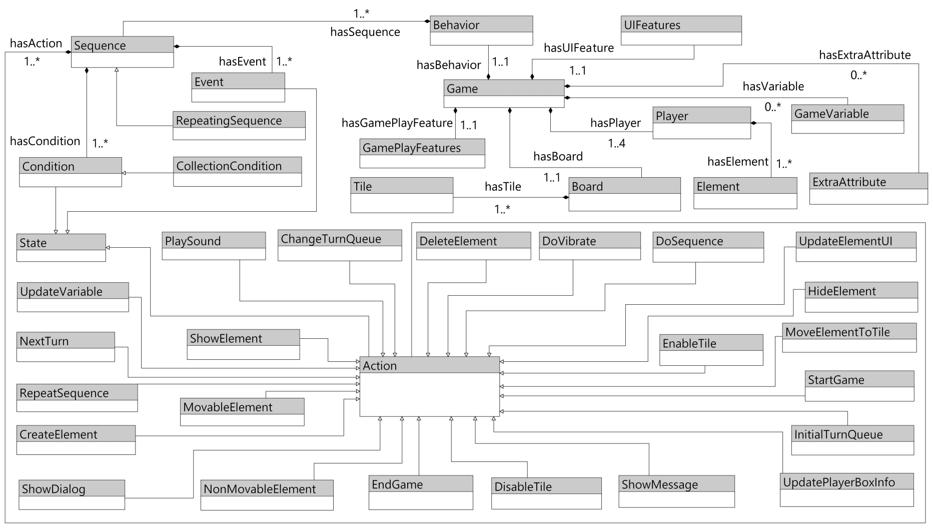


Fig. 6: Excerpt of the Game Structure and Behavior Metamodel

GamePage have to be drawn completely independent from other pages and in a dynamic way using the graphic libraries of Android. However, the user interface in the game menu pages are of a static form based on XML files. To be able to create diverse *GamePage* pages, some features of this page are abstracted and are placed in the ***UIFeatures*** component. This enables the user to personalize the game page display by allocating desired quantities to these features. For instance, Figure 7 represents the general pattern designed for the *GamePage* in MAndroid. The design is made up of three parts or panels. *Top Panel* contains the menu button and two spaces for displaying information of the two players. These spaces are provided depending on the number of players playing the game, and are called *Player Box*. The middle panel, declared as the *Board Panel*, contains the required space to place the board component. The *Bottom Panel* can contain the throw dice button and two other spaces to display the information of two additional players, depending on the number of players and game style. Every *Player Box* contains two different spaces to accommodate desired texts, which might be the player's name, points, turns, etc. This space is separated by a line that passes through the box. Every *Player Box* is made up of an internal and an external space, which may have different backgrounds.

Despite the board view being square in MAndroid, we can model games that have a non-square screen. In fact, the board in MAndroid is just a space for placing tiles, and the arrangement and placement of the tiles determines the appearance of the game screen, not particularly the board itself. Figure 8 presents the Ludo game, whose board never has a grid view, however, it is trivial to model it in MAndroid. According to Figure 9, it is still possible to

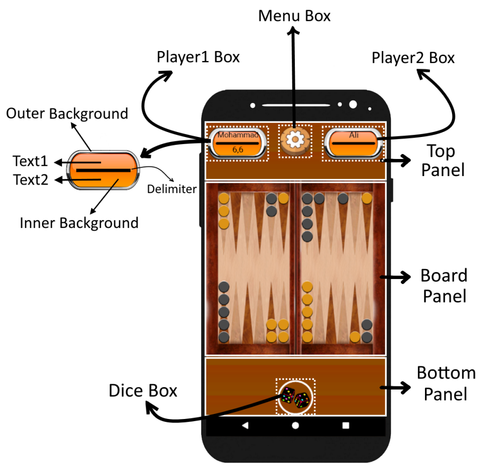


Fig. 7: The general pattern for the GamePage in MAndroid

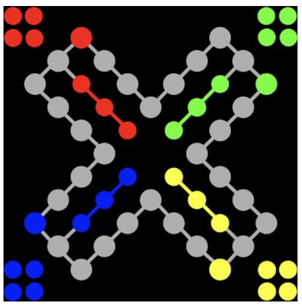


Fig. 8: The board of Ludo game

consider it on a square screen, and disable extra tiles. To disable additional tiles, it is required to set their *inEnable* attribute to *false*. Figure 10 shows the inactive tiles of the game, which are displayed in white, while the active tiles have a black background.

Like in the previous metamodel, the *Game* contains ***GameVariables*** to declare further variables. Obviously, a *Game* has a ***Board***, which is the place that accommodates all the components of the game. The *Board* is made up of ***Tile***s organized as a matrix (cf. Figure 9). These *Tile*s are locations in which game elements can be allocated. Since the board has to be a rectangular or square space, every *Tile* can be easily identified using its row and column index number. The numbering of rows and columns is started from the top-left corner with zero. The attribute number (cf. Figure 22) is used to calculate the row and column of the *Tile*, which makes it simpler for the user.

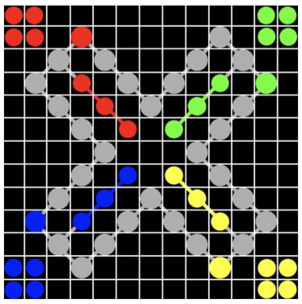


Fig. 9: The Ludo game in square board

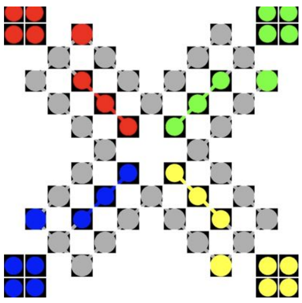


Fig. 10: The Ludo game in square board with Deactivated tiles in white color

We consider that a *Game* is made up of up to four ***Player***s, since we are considering multiplayer games. Every *Player* has a number of ***Element***s, which model the progression in a board game. Every *Element* must be in a *Tile*, moves between Tiles, gets visible, hides, is added and removed from the board, etc.—attributes of *Element* are shown in Figure 22. As an example, in a chess game, *Element*s are the pawn, knight, bishop, etc. Generally, the structure of board games in MAndroid is formed by the board, tiles and the elements. Figure 11 shows the Othello game. In the figure, the board is made up of 8 columns, 8 rows and therefore 64 tiles.

The most challenging part in designing a game is defining its rules, which is defined by its ***Behavior***. UML structural diagrams such as class diagrams do not provide all necessary features to model the behavior and rules of the game. Therefore it is necessary to use UML behavioral diagrams, such as state machines. For this purpose, a new method has been proposed in the MAndroid framework that combines the concepts used in class diagrams and state machines in order to model the game behavior using Ecore diagrams. The game behavior in MAndroid is made up of three components including ***Condition***, ***Action*** and ***Event***. *Event*s are executed by the *Player* during the game. For example, throwing the dice is an *Event* that is called by pressing on the dice icon. If an *Event* is reached during the game flow, the flow will

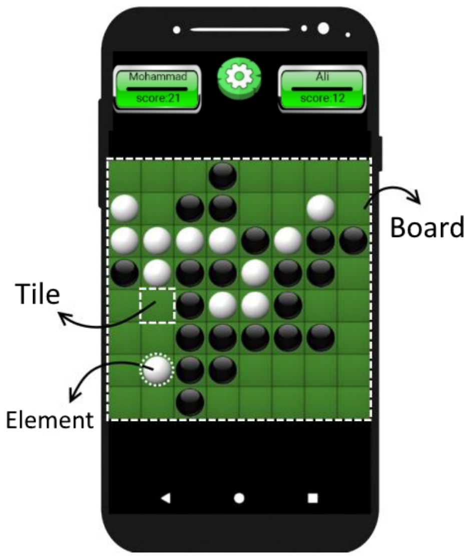


Fig. 11: Structure of the Othello game in MAndroid

be stopped until the *Event* is finished. *Condition* is another component of the game used to check the correctness of statements and, in turn, to decide on the continuation of the game. For example, after an *Action* by the user, in case the element lands on a *Tile* with even number, the game will continue on path A; and if the element lands on an odd number tile, it will continue on path B. *Action* is a component used to perform several actions that have been defined under the domain of the game. For example, deleting an element from the board when hit by an enemy element is an *Action*. Figure 12 displays the principles of behavioral modeling in MAndroid. According to the figure, the game starts at a singular point and ends in another point. After the game starts, at every point of time, it is at a specific point called ***State***, which can be either an *Event*, *Action* or *Condition*. A concept named ***Sequence*** is used to ease the modeling and prevent the implementation of repetitive *State*s. Every *Sequence* is made up of a set of *State*s and does not perform any change in the game by itself; instead, it is used to organize and keep in order the *State*s to be used. In case the need to use a *Sequence* more than once arises, the ***RepeatingSequence*** component can be used. Figure 12 shows a game that uses 3 *Sequences* and 12 different *States*. The game enters a *State* in the form of an *Event*. The flow of the game stops until the intended *Event* is called by the user. Then the game reaches a *State* or *Condition* where the game can be either directed to the right or left. Finally, if the game is directed towards its ending point, it will end. All behavioral states and rules of the game can be modeled using this method.

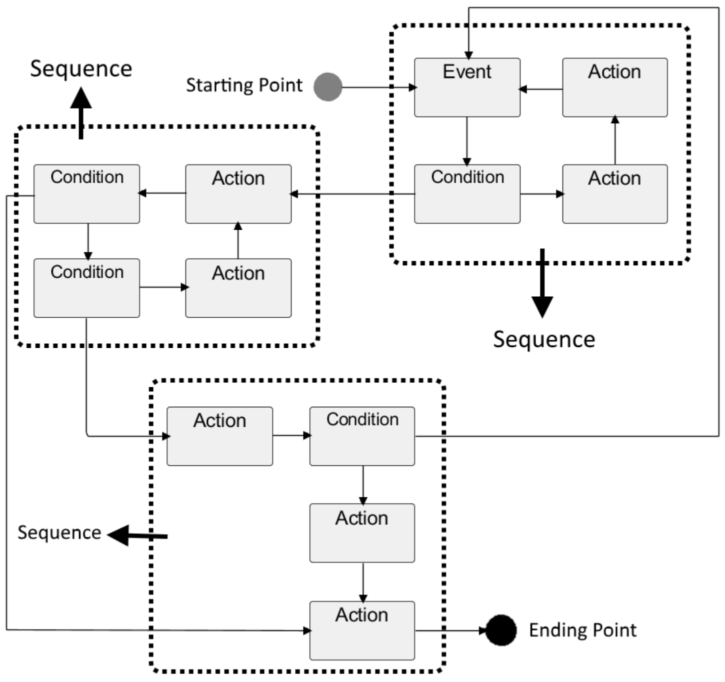


Fig. 12: Principles of behavioral modeling in MAndroid

*Action*s are used to undertake an action in the game. The actions have been designed by considering the domain of classic 2D board games, so that that all needed actions are possible. For instance, ***StartGame*** is used to start the game showing a welcome message to the users, ***NextTurn*** shifts the turn to the next player in the queue, ***MoveElementToTile*** is used to move an element from a *Tile* to another, and ***ShowDialog*** and ***ShowMessage*** are in charge of displaying dialogs and messages to the user (cf. Figure 13). All *Action*s and a detailed description of all components in the metamodel, together with the full metamodel, are available in Appendix A.

4.2 Modeling in MAndroid

Since our approach provides the menu and structure and behavior metamodels, users of our framework can define models that conform to these metamodels. Different tools available in the Eclipse framework can be used to define these models. We use the Exeed editor, which is a built-in graphical editor for the Eclipse framework based on Ecore. It allows to build models using a tree pattern. In MAdroid, users need to define a model that conforms to the menu metamodel and another model that conforms to the structure and behavior

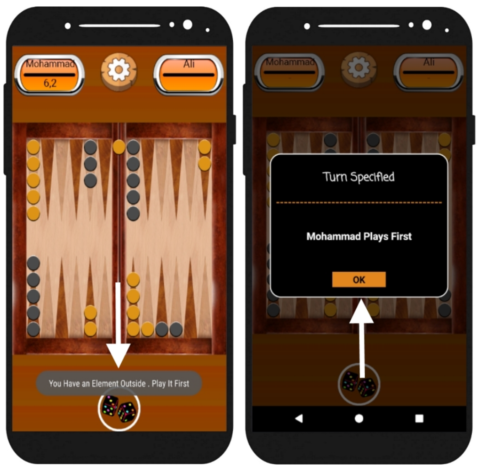


Fig. 13: Example of a dialog window and a message window in MAndroid

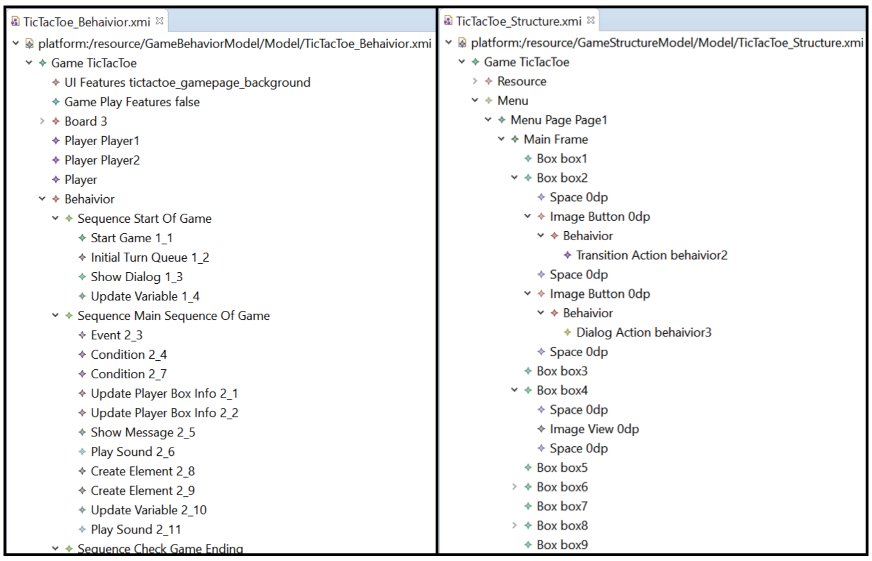
metamodel. Both models serve as input to the transformation engine that generates the final code.

Figure 14 displays a model conforming to the structure and behavior metamodel in the left-hand side, and a model conformig to the menu metamodel in the right-hand side. We can see that the top-most component in both models is the *Game* component (in this case we are modeling the Tic-Tac-Toe game). All remaining components are added following the structure provided by the metamodels.

4.3 Model Transformation in MAndroid

The last step is to apply the model-to-text transformation that generates the final code. For this, the models conforming to the metamodels described in the previous section serve as input. The Acceleo transformation language has been used for implementing the transformation. The model-to-text transformation is divided in two main modules. First, the menu model serves as input of the first main module of the transformation, and the *AndroidManifest.xml* file is generated, in which essential variables are declared. Then, the model conforming to the structure and behavior metamodel is used as input for the second main module, whose output is a set of Java classes implementing the logical functioning of the game. The model transformations are available from [45].

The XML files and Java classes generated by the transformation need to be transferred to the pre-generated project in Android Studio, which contains the static parts of the game. Figure 15 presents the generated Java files for the Tic-Tac-Toe game in the MAndroid framework after applying the transformations.



```
TicTacToe_Behaivior.xmi
platform:/resource/GameBehaviorModel/Model/TicTacToe_Behaivior.xmi
  Game TicTacToe
    UI Features tictactoe_gamepage_background
    Game Play Features false
    Board 3
    Player Player1
    Player Player2
    Player
    Behaivior
      Sequence Start Of Game
        Start Game 1_1
        Initial Turn Queue 1_2
        Show Dialog 1_3
        Update Variable 1_4
      Sequence Main Sequence Of Game
        Event 2_3
        Condition 2_4
        Condition 2_7
        Update Player Box Info 2_1
        Update Player Box Info 2_2
        Show Message 2_5
        Play Sound 2_6
        Create Element 2_8
        Create Element 2_9
        Update Variable 2_10
        Play Sound 2_11
      Sequence Check Game Ending
```

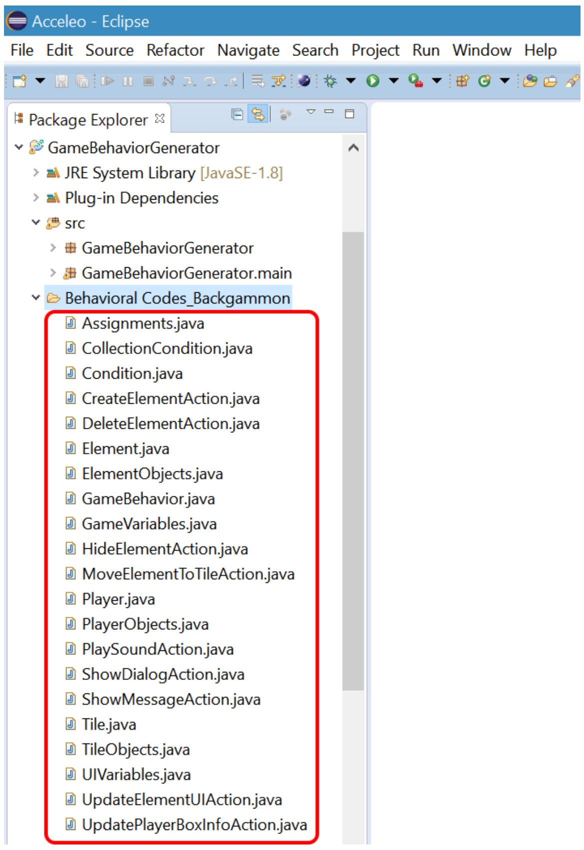
```
TicTacToe_Structure.xmi
platform:/resource/GameStructureModel/Model/TicTacToe_Structure.xmi
  Game TicTacToe
    Resource
    Menu
      Menu Page Page1
        Main Frame
          Box box1
          Box box2
            Space 0dp
            Image Button 0dp
              Behaivior
                Transition Action behaivior2
            Space 0dp
            Image Button 0dp
              Behaivior
                Dialog Action behaivior3
            Space 0dp
          Box box3
          Box box4
            Space 0dp
            Image View 0dp
            Space 0dp
          Box box5
          Box box6
          Box box7
          Box box8
          Box box9
```

Fig. 14: Menu, structure and behavior modeling of a board game in MAndroid

5 Evaluation

For the evaluation of our approach and framework, we have taken into consideration three classic multiplayer 2D board games, namely Backgamon, Othello and Tic-Tac-Toe. Out of the three categories of classic board games described in Section 2.1.2, Backgammon is a race game, Othello is a war game and Tic-Tac-Toe is an alignment game. As we explain later, we are confident these three games cover all possible structural and behavioral features of classic multiplayer 2D board games, so implementing these games in MAndroid proves the applicability of our framework. We also organized a workshop with 34 attendees who are graduate and undergraduate students in Computer Engineering. The purpose of this workshop was to develop the Tic-Tac-Toe game using MAndroid. Tic-Tac-Toe was selected because it is a popular board game, so most people are familiar with the game and its rules. Besides, this game can be developed within four hours, which is the time taken by the workshop. After the workshop, students were asked to fill out a couple of questionnaires with a comprehensive set of questions related to the use of MAndroid.

Additionally, in order to compare MAndroid with other game development environments, 30 developers with expertise in Android game development were asked to fill out another couple of questionnaires. It was not trivial to find developers who were expert in developing multiplayer 2D board games, therefore we only selected participants with experience in game development in Android. As in the workshop, we limited the focus of these questionnaires to the Tic-


Acceleo - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer

GameBehaviorGenerator
JRE System Library [JavaSE-1.8]
Plug-in Dependencies
src
GameBehaviorGenerator
GameBehaviorGenerator.main
Behavioral Codes_Backgammon
Assignments.java
CollectionCondition.java
Condition.java
CreateElementAction.java
DeleteElementAction.java
Element.java
ElementObjects.java
GameBehavior.java
GameVariables.java
HideElementAction.java
MoveElementToTileAction.java
Player.java
PlayerObjects.java
PlaySoundAction.java
ShowDialogAction.java
ShowMessageAction.java
Tile.java
TileObjects.java
UIVariables.java
UpdateElementUIAction.java
UpdatePlayerBoxInfoAction.java

Fig. 15: Generated code of the Backgammon game in the MAndroid framework

Tac-Toe game. This helped us compare MAndroid with existing frameworks for Android game development.

Finally, some baseline implementations of the Tic-Tac-Toe game were selected from Google Play for comparison with the version obtained with MAndroid. It is not trivial to find the games with identical features to the generated game in this research. However, the games with more identical features have been selected. We shall recall that the focus of this paper is on classic multi-player 2D board games, so games such as Monopoly, where cards are used, or games like Carcassonne, where the board construction is an integrated part of the game, are not considered in this evaluation. In the remaining of this section, we present the research questions and the experimental setup of our evaluation.

5.1 Research Questions

The research questions (RQs) that we want to answer in this paper and the rationale for each RQ are the following:

- ***RQ1 - Applicability.*** *Is it possible to build other classic multiplayer 2D board games using MAndroid? Is it possible to extend the MAndroid framework for other game styles and platforms?* Since our MAndroid proposal covers all behavioral and structural features of the games, we want to discover whether it would be possible to develop the structural and behavioral features of any classic multiplayer 2D board game using MAndroid. Additionally, we want to determine whether it would be possible to extend the MAndroid approach and framework for developing other styles of games and with support for other mobile platforms.
- ***RQ2 - Development Performance.*** *How fast can a developer implement a game using MAndroid?* We specially focus on learning time, game development time and time for discovering and solving errors found during development. These three aspects are stated as sub-questions:
  - ***RQ2.1 - Learning Time.*** *To what extend does game development with MAndroid reduce learning time?* This learning time involves principles and concepts needed for game development.
  - ***RQ2.2 - Development Time.*** *To what extend does game development with MAndroid reduce development time?* To answer this, we measure the development time[4] of the Tic-Tac-Toe game in the workshop (cf. Section 5.2.2) from beginning to end.
  - ***RQ2.3 - Time for Discovering and Solving Errors.*** *Is our framework able to reduce the time for discovering and solving semantic errors?* The amount of time for solving particular issues in the process of game development with MAndroid is compared to other approaches and frameworks.
- ***RQ3 - Simplicity.*** *How simple is the development of games using MAndroid?* We want to investigate to what extend the level of knowledge of the participants in software engineering, MDE, Android and game development influence the simplicity of the approach. Additionally, the number of questions that may raise during game development, i.e. how often the workshop participants approached the organizers to ask questions related to the framework, is used as an indicator of the simplicity of the approach.
- ***RQ4 - Attractiveness.*** *How attractive is the MAndroid framework?* Having workshop participants from different levels and expertise, we want to study the evaluation of the attractiveness of the framework/approach after developing the game. Questions 11, 13 and 14 in Table 4 are used to evaluate this RQ. In particular, we measure how attracted the user is, after attending the workshop, to (i) MAndroid, (ii) MDE methodology and (iii) game development.

[4] By *development time* in MAndroid, we really refer to *modeling time*

– ***RQ5 - Technical Evaluation.*** *How does the game generated with MAndroid result in terms of technical criteria—such as number of generated APK files, memory occupation and CPU usage—compared to similar games?* We want to compare the Tic-Tac-Toe game that is generated using the MAndroid framework to other Tic-Tac-Toe implementations available in the market, which were developed using different frameworks. The compared games may be similar in terms of general game features and style, but are likely to have some technical differences. The latter are subject of study in this RQ.

5.2 Experimental Setup

5.2.1 Case studies

Three games are used as case studies, namely Backgammon, Othello and Tic-Tac-Toe. Despite questionnaires (cf. Sections 5.2.2 and 5.2.3) are focused on the Tic-Tac-Toe game, the other two games have also been implemented using MAndroid, which helps demonstrate its applicability. Figure 16 displays the interfaces of the three games as generated with MAndroid. The description of these games are as follows:

– **Backgammon** is one of the oldest well-known board games in the community. In this game, the turn of players is identified by rolling a die and the player with the higher roll starts the game. A player is allowed to roll both dice in each turn. If the same number shows on both dice, the player is able to play again with the same number. It is a two-player game in which the first player moves counterclockwise from the upper right, while the opponent's moves are clockwise from the bottom right. It is also possible to change the direction. In this game each player has 15 pieces that can be moved between twenty-four triangles of the board. The winner of the game is the one who first moves all 15 pieces off the board. For each move, a player moves the piece off a triangle on the board that contains no pieces, the player's own pieces or a single opposing piece. The opposing piece gets removed if other player moves a piece onto that point. In this case the owner of the piece requires to return it back to the board and starts from the beginning. In this game the player starts the process of removing pieces from the board, after all pieces are moved on to the board. Each piece is removed if the player rolls a number corresponds to the specific point that piece is resisted. If there is no piece on that point then the player is able to remove other piece with less value than the value presented on the die. The winner of this game is the player who bears off all the pieces first [46].
– **Otello** is an alternative board game, in which a color is assigned to each player. In this game the rule is to capture opponents' pieces and flip them over to turn them to your color. It is possible to have a horizontal, vertical or diagonal line of pieces to enclose opponents' pieces. The game is finished

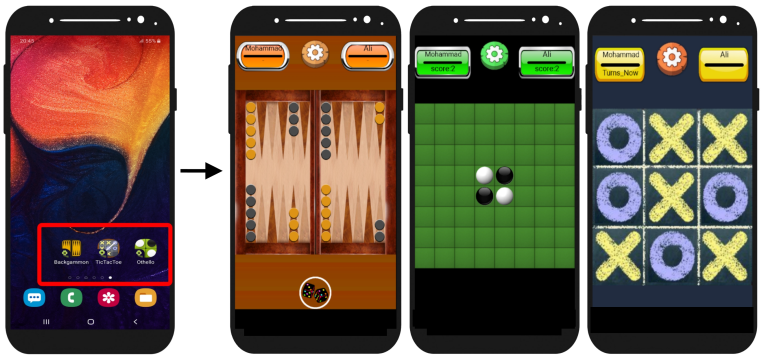


Fig. 16: The Backgammon, Othello and Tic-Tac-Toe games developed by MAndroid

Table 2: Structural features of classic board games

Structural Feature	**Chess**	**Backgammon**	**Othello**	**Ludo**	**Snakes -Ladders**	**Tic-Tac-Toe**	**Checkers**	**Go**
Board	✓	✓	✓	✓	✓	✓	✓	✓
Tile	✓	✓	✓	✓	✓	✓	✓	✓
Element	✓	✓	✓	✓	✓	✓	✓	✓
Dice		✓		✓	✓			
Two/More players	✓	✓	✓	✓	✓	✓	✓	✓

when the board is full. The winner of this game is the one who gets more pieces on the board [47].

– **Tic-Tac-Toe** is a game for two players, one with X pieces and the other with O pieces. The rule of this game is to arrange 3 pieces in a row. Players place their pieces (X or O) on a 3x3 square board. The winner is the one who first places 3 pieces vertically, horizontally, or diagonally. The players tie if the board is full and none of the participants manages to generate 3 of their pieces in a row [46].

Tables 2 and 3 horizontally present all structural and behavioral features, respectively, that need to be taken into account for developing classic multiplayer 2D board games in Android. In Table 3 arbitrary, game features are those features that are not essential for developing the game. For instance, audio or vibration are arbitrary features of the game and it is still possible to run the game without these features. On the contrary, compulsory game features are essential for the game and without these features the game is not completed. For instance, moving the elements is essential in Chess game, but not compulsory for Tic-Tac-Toe game as the elements of this game can

Table 3: Behavioral features of classic board games

Behavior Feature	**Chess**	**Backgammon**	**Othello**	**Ludo**	**Snakes -Ladders**	**Tic-Tac-Toe**	**Checkers**	**Go**
Moving element	✓	✓		✓	✓			✓
Hitting element	✓	✓		✓	✓			
Creating new element	✓		✓			✓	✓	✓
Hiding the element	☑	☑	☑	☑	☑	☑	☑	☑
Changing element's appearance	☑	☑	☑	☑	☑	☑	☑	☑
Throwing the dice		✓		✓	✓			
Changing turn	✓	✓	✓	✓	✓	✓	✓	✓
Increase or decrease player points			✓					
Performing process on touched elements	☑	☑		☑	☑		☑	
Performing process on clicked elements	☑	☑		☑	☑		☑	
Performing process on moved elements	✓	✓		✓	✓		✓	
Performing process on clicked Tile	☑	☑	✓	☑	☑	✓	☑	✓
Performing process on the value of throwing dices		✓		✓	✓			
Presentation of name and points of players	☑	☑	☑	☑	☑	☑	☑	☑
Showing message to players	☑	☑	☑	☑	☑	☑	☑	☑
Playing sound and vibration	☑	☑	☑	☑	☑	☑	☑	☑

✓Compulsory game feature. ☑Arbitrary game feature.

not be moved after placing them on the game board. The columns represent how many of these features are covered by seven different games. Having a look at Table 3, we can see that there is no game in this domain that supports all behavioral features. We can also see that the three games developed with MAndroid—Othello, Backgammon and Tic-Tac-Toe—cover together all structural and behavioral features.

As mentioned, for the purpose of the evaluation, in the questionnaires we selected the Tic-Tac-Toe game. We can see in Tables 2 and 3 that this game covers most of the structural and behavioral features. It is assumed most of the workshop's participants were familiar with this game and it would be possible to implement the rules and constraints of this game in the time allocated for the workshop.

5.2.2 Workshop and Participants

A practical workshop was organised in the Department of Computer Engineering at the University of Isfahan. In this workshop, 4 PhD, 5 Master and 15 Undergraduate students participated. A master student and a PhD student were expert in the domain of MDE and had previous experience in generating domain-specific modeling languages with MDE concepts. The remaining graduate participants were familiar with MDE concepts from the material of a Model-Driven Software Engineering course, although they had not developed any practical project using MDE. All selected undergraduate students were in the last year of their degree and ranked top-10 scores during their studies. They were all familiar with software engineering and programming concepts, and they had different expertise. The purpose of the workshop was to develop the Tic-Tac-Toe game using the MAndroid framework. Before the workshop, participants were provided with four videos (they are available from `https://shorturl.at/evzFW`[5]):

- The first video explains the Tic-Tac-Toe game and its rules. To have consistent implementations of the game, its requirements for the purpose of the workshop were explained.
- The second video introduces the general concepts of this research, including MDE, model transformations, metamodeling, modeling and board games. Additionally, the introduction to the structure of board games was provided in this video.
- The third video explains the behavior of board games.
- The forth video describes EMF and Ecore concepts and provides an introduction to the creation of the project, metamodels and models.

The workshop took 4 hours, including 90 minutes for training at the beginning of the workshop. During the training part, the same materials that had been given to the participants before the workshop in video format were explained to all participants. This helped ensure that everything was understood by everyone.

Some participants managed to finish the implementation of the game in less than 4 hours. The behavior of the participants during the workshop was analysed. Development time, number of questions asked during the workshop and the number of errors during implementation and time for solving them were recorded. Participants also filled out a couple of questionnaires after the workshop based on their experience on modeling and developing the Tic-Tac-Toe game with MAndroid. The answers in the questionnaire forms were used to evaluate the MAndroid approach from different perspectives.

Additionally, in order to compare MAndroid with other game development frameworks (code-centred, game engines, game creation tools), another couple of questionnaires were designed. To fill them out, 30 experts in Android game development were selected. As in the workshop explained above, the Tic-Tac-Toe game was selected as classic multiplayer 2D board game under study. The requirements of this game were explained to the experts to have a consistent idea of the structural and behavioral features of the game. They were also provided with screenshots of the Tic-Tac-Toe game. The answers to these questionnaires were used to compare MAndroid with other game development frameworks. All four questionnaires are detailed next.

[5] Please note, videos are in Farsi

5.2.3 Questionnaires

A comprehensive set of questions were asked in the questionnaire forms to provide appropriate answers to the RQs (cf. Section 5.1). Tables 4 and 5 present the questionnaire forms of the workshop (cf. Section 5.2.2). Answers to questions in the first questionnaire were encoded using a Likert scale [48] with values Very much, Much, Average, Low or Very Low. The main advantage of using a Likert scale is that the questions involve use a similar method of collecting the data, which makes them easy to understand and answer, and students do not feel forced to express their opinion, allowing them to stay neutral. Additionally, the use of a Likert scale eases the joint analysis of the questionnaires.

In Table 4, questions 1 to 5 relate to the knowledge of the workshop's participants. We asked about the participants knowledge in different aspects, namely software modeling and MDE, Android development, 2D game development, EMF and Ecore. In questions 6 and 7, the applicability of MAndroid for the development of structural and behavioral features of the Tic-Tac-Toe game was asked (RQ1). Also regarding applicability, questions 8 and 9 correspond to the applicability of MAndroid in developing board games, and applicability is further measured with the replies to questions 15 to 18. Question 10 is used for evaluating the simplicity of MAndroid (RQ3). Additionally, the results of the workshop are used for measuring this criteria. Questions 11, 13 and 14 evaluate the attractiveness of MAndroid (RQ4). Question 12 relates to the developer performance and measures the time for solving errors in MAndroid (RQ2.3). The results of the workshop were also used for evaluating this criteria. Question number 19 in the second questionnaire asked for the time needed for learning provided materials and concepts before the workshop (RQ2.1). The second column of Table 4 presents the relation between the questions in the questionnaires and the RQs formulated in Section 5.1. Apart from these two questionnaires, participants' behavior in terms of time for watching video tutorials, development time of the game, number of questions made during development, number of semantic errors discovered and time for solving and discovering these semantic errors were recorded during the workshop (RQ2, cf. Table 9 and Section 6.2).

Tables 6 and 7 present the questions corresponding to the experts' questionnaires (cf. Section 5.2.2). It was not possible to organize a practical workshop for this part of the evaluation as the participants would require different platforms and environments for developing the game, so all questions were based on the experts' opinions regarding the development framework in which they have expertise. We provided the participants with the structural and behavioral requirements of Tic-Tac-Toe using the screenshots of different stages of the game. It was assumed that all resources such as images and sounds would be provided for developing the game. Seven questions were asked, from which three questions were related to the required level of knowledge of developers of Android, game development, 2D graphics and modeling with UML or other

Table 4: First workshop questionnaire

Number	RQ	Question	Very much	Much	Average	Low	Very Low
1	3	To what extend are you familiar with software modeling and MDE?					
2	3	What is your level of knowledge in Android development?					
3	3	What is your level of knowledge in 2D game development?					
4	3	To what extend are you familiar with EMF?					
5	3	To what extend are you familiar with Ecore?					
6	1	To what extend could you manage to model the structure of the Tic-Tac-Toe game?					
7	1	To what extend could you manage to model the behavior and rules of the Tic-Tac-Toe game?					
8	1	How applicable is MAndroid in the development of the structure of board games?					
9	1	How applicable is MAndroid in the development of the behavior of board games?					
10	3	How simple and understandable is the development of the game using MAndroid?					
11	4	How attractive is MAndroid for developing the game?					
12	2,3	How effective is MAndroid in detecting and solving semantic errors in comparison to code-centric approaches?					
13	4	To what extend are you attracted to MDE methodology after attending this workshop?					
14	4	To what extend are you attracted to game development after attending this workshop?					
15	1	To what extend is MAndroid applicable in the development of other 2D board games?					
16	1	To what extend is MAndroid extensible in the development of other styles of 2D games?					
17	1	To what extend is MAndroid applicable in the development of other classic multiplayer 2D games?					
18	1	To what extend is MAndroid applicable in the development of 2D board games for other mobile platforms?					

MDE approaches (Table 7). Answers to these questions were encoded using a Likert scale [48] with values Very much, Much, Average, Low or Very Low.

Four questions were asked in the other set of questions (Table 6). The first one was about the time that would be required to develop the Tic-Tac-Toe game in the corresponding platform in which the respondent has expertise. The second question was about the required time for learning different concepts for developing the game according to its requirements. Question three was about the number of questions that experts think they would need to perform while developing the game. These refer to aspects that could not be solved based

Table 5: Second workshop questionnaire

Number	**RQ**	**Question**	**Answer**
19	2	How much time did you spend before the workshop in learning MAndriod concepts, using the provided video tutorials?	

Table 6: First experts questionnaire

Number	RQ	**Question**	**Answer**
1	2	How much time do you think is it required to develop the Tic-Tac-Toe game based on the specified requirements?	
2	2	How much time do you think it is required to learn different concepts for developing the Tic-Tac-Toe game based on the specified requirements?	
3	3	Since you might face problems while developing the game, how many questions do you think you would need to ask or how many resources do you think you would need to check in order to find the answer to your questions?	
4	2	How much time do you think you would need in order to to solve semantic errors during development of the game?	

Table 7: Second experts questionnaire

Number	RQ	**Question**	**Very much**	**Much**	**Average**	**Low**	**Very Low**
5	3	How much expertise in Android development is required to develop this game?					
6	3	How much expertise in game development and 2D graphics are required to develop this game?					
7	3	How much expertise in modeling with UML or other MDE approaches are required to develop this game?					

only on experts' knowledge. Finally, the time for solving hypothetical semantic errors during game development from experts' perspective was asked. The idea with these questions is to compare experts' opinions regarding developing the Tic-Tac-Toe game using the platform in which they are expert with the answers from the participants in the workshop after developing the game with MAndroid (cf. Section 5.2.2). The second column of tables 6 and 7 present the relation between these seven questions and the research questions, where we see that they help answer RQs one to three.

5.2.4 Baseline Implementations

For a technical evaluation of the Tic-Tac-Toe game developed with MAndroid, it is compared to seven other implementations available in the market. The games are selected from Google Play, and they are highly similar to the game obtained with MAndroid. The popularity and rating of the games were

Table 8: Tic-Tac-Toe game implementations available on Google Play

Game name	**Developer**	**Number of installations**	**Score (0-5)**	**Latest update**	**Package**
Tic Tac Toe	Wintrino [49]	50,000,000+	4.3	December 18 2018	com.tictactoe.w+
Tic Tac Toe glow - Free Puzzle Game	Arclite [50] Systems	10,000,000+	3.9	May 20 2019	com.arcsys.tict+
Tic Tac Toe Classic	AlmaTime	10,000,000+	4.1	February 21 2020	com.almatime.ti+
Tic Tac Toe 2	BYRIL [51]	1,000,000+	4.3	November 23 2018	com.byril.ticta+
Tic Tac Toe	Fun Games free [52]	1,000,000+	3.5	December 18 2018	tic.tac.toe.gem+
Tic Tac Toe	Big Brain Kraken [53]	1,000,000+	4.3	December 12 2019	com.bigbrainkra+
Tic Tac Toe Glow: Multiplayer!	Playzio [54]	100,000+	3.4	September 14 2017	com.playzio.tic+

also considered. Table 8 presents information about these games. Later (cf. Section 6.5), the number of APK files generated, the percentage of memory occupation and the percentage of CPU usage are used for comparing the Tic-Tac-Toe game generated from MAndroid with these seven games. This allowed answering RQ5.

6 Results

This section is devoted to answer all RQs formulated in Section 5.1. The answers to the questions encoded using a Likert scale that workshop participants submitted after the workshop (cf. Section 5.2.3 and Table 4) are displayed in Figure 17. Regarding experts' answers to the questions using a Likert scale (cf. Section 5.2.3 and Table 7), they are displayed in Figure 18.

Correlation measures monotonic association between variables [55]. This means a change in one variable may affect another variable in a positive or negative way. In order to analyse the correlation between the different answers in this research, the Spearman's correlation is used to calculate the linear relationship between variables [56]. A positive relationship between the metrics implies a positive correlation, while a negative relation indicates a negative correlation. The closer the coefficient is to either -1 or 1, the stronger the correlation between the variables is. The strength of the correlation is classified into five different ranges. The values in the range [0, 0.19] are very weak, in [0.20, 0.39] are weak, in [0.40, 0.59] are moderate, in [0.60, 0.79] are strong and in [0.80, 1.0] are very strong.

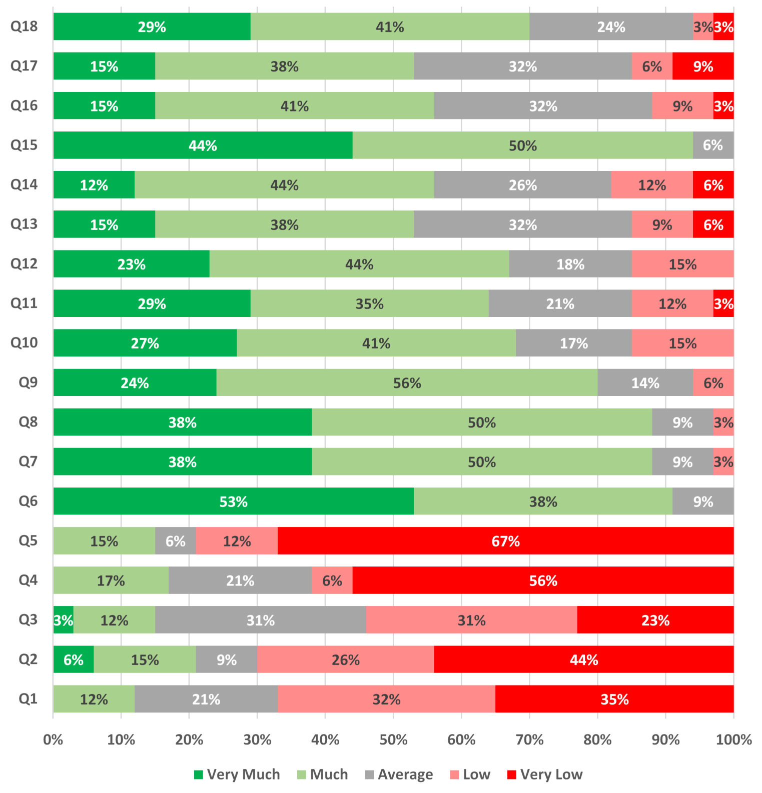


Fig. 17: Responses of workshop participants to the questions in Table 4

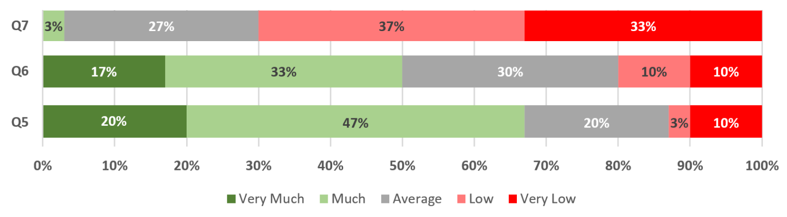


Fig. 18: Experts' responses to the questions in Table 7

6.1 RQ1- Applicability

The applicability of our MAndroid approach and framework is evaluated both theoretically and practically. Having a look at the structural and behavioral features of the Tic-Tac-Toe, Othello and Backgammon games in Tables 2 and 3, we can appreciate the applicability of MAndroid in modeling and developing all possible structural and behavioral features of multiplayer 2D board games.

Therefore, the provided metamodels are complete and all the details for developing games in this domain are considered. Additionally, adequate sets of model-to-text transformations have been implemented in the approach, which allow to generate the games completely and automatically.

According to the answers of workshop participants to questions 6 and 7 (cf. Table 4 and Figure 17), 91% of participants believed they had to a very large extent (53%) or to a large extent (38%) modeled the game structure, while 88% of participants believed they had to a very large extent (38%) or to a large extent (50%) modeled the game behavior. This goes in accordance with the results analyzed after the workshop, where we observed that around 95% of game structure and 84% of game behavior had been successfully implemented in the workshop. Additionally (questions 8 and 9), 88% of users definitely approved (38%) or approved (50%) the applicability of MAndroid in the development of the structure of board games, while 79% definitely approved (23%) or approved (56%) its applicability in the development of the behavioral part. There are strong positive linear correlations among the answers to questions 6 to 9 of the questionnaire, what indicate that participants did not reply randomly.

The extensibility of the MAndroid approach is evaluated with questions 15 to 18. The results in Figure 17 show that only 12% of participants believe in a low applicability of the MAndroid framework in terms of extensibility for other styles of 2D games. This is 15% for multiplayer games and 6% for other mobile platforms. A positive linear moderate and strong correlation results between answers to questions 15 to 18 of questionnaire are reported. Therefore, according to workshop participants, it can be concluded with confidence that the MAdroid approach is extensible and it is possible to apply it to other styles of 2D games and multiplayer games in other mobile platforms.

6.2 RQ2 - Developer Performance

Developer performance using MAndroid is evaluated in terms of learning time of related concepts, development time of the Tic-Tac-Toe game and time for solving semantic errors found during development. To measure this criteria, we recorded participants' interactions and results in the workshop.

The video watching time is the time each participant spent in investigating and watching the provided materials before the workshop, which is the response to Q19 in Table 5. Second column of Table 9 shows the watching time for each participant. The time that each participant spent in modeling and generating the Tic-Tac-Toe game using our MAndroid framework is displayed in the third column. Participants also needed to approach workshop organizers to ask some questions about the development. The number of questions asked by the participants is shown in the fourth column. After participants finished the modeling phase, MAndroid executes the model-to-text transformations and generates the corresponding Android implementation. The generated code was run on an Android emulator and, if there was any problem, the errors were

Table 9: Participants behavior in the workshop

ID	**Video watching time**	**Development time**	**# Questions**	**# Errors**	**Time for solving errors**	**Time for solving each error**
1	30	65	5	3	36	12
2	40	123	2	1	22	22
3	20	140	4	NA	NA	NA
4	30	166	2	2	40	20
5	20	105	3	1	36	36
6	15	90	4	2	39	19.5
7	30	122	2	2	22	11
8	40	115	2	4	48	12
9	45	90	1	1	13	13
10	15	74	1	NA	NA	NA
11	0	87	3	2	40	20
12	0	90	5	NA	NA	NA
13	0	90	6	2	35	17.5
14	60	90	0	1	4	4
15	0	120	3	1	10	10
16	15	103	1	1	9	9
17	30	90	0	1	27	27
18	0	114	3	2	35	17.5
19	0	87	2	NA	NA	NA
20	45	90	0	2	21	10.5
21	0	90	1	1	10	10
22	30	125	4	1	13	13
23	0	90	0	NA	NA	NA
24	30	132	2	4	35	8.75
25	0	105	3	2	15	7.5
26	0	80	1	3	45	15
27	0	105	4	1	20	20
28	15	112	3	1	18	18
29	0	145	2	0	0	0
30	30	75	5	2	20	10
31	30	68	6	1	10	10
32	0	134	2	4	60	15
33	10	170	3	2	46	23
34	10	60	1	1	9	9
AVG	17.35	104.17	2.52	1.75	25.44	14.49

displayed. The number of errors obtained and time for solving them are shown in the fifth and sixth columns of the table, while the last column displays the time for solving each error. Please note that five participants (with IDs 3, 10, 12, 19, 23) did not want to continue after the modeling phase, therefore the number of errors and time for solving those errors were not recorded for them. The reason was these participants felt tired and did not want to continue with the implementation. For those participants, we included *NotAvailable* (NA) in the corresponding table cell, and the value of such cells was not considered in the analysis.

Regarding experts' opinions on these matters under a hypothetical implementation of the Tic-Tac-Toe game using their platform of expertise, questions were asked in the questionnaire of Table 6 (cf. Section 5.2.3). The answers are presented in Table 10. With all this we can answer the three sub-questions.

We shall highlight that comparing the time spent by workshop participants with the time experts think they would spend does not seem fair. However, as mentioned earlier, it was very hard to prepare a workshop for experts to

Table 10: Evaluation results for questionnaires of Android experts

ID	Concepts learning time	Development time	# Questions	Time for solving errors
1	40	24	0	60
2	20	60	0	15
3	100	120	1	5
4	75	240	1	5
5	5	480	2	5
6	20	480	12	10
7	4	600	2	10
8	24	600	2	20
9	100	720	2	120
10	20	720	4	20
11	10	900	5	20
12	100	900	5	20
13	25	1050	5	20
14	24	960	10	20
15	40	1200	10	20
16	35	1500	10	25
17	30	1500	15	25
18	100	1800	3	25
19	50	270	4	30
20	40	2400	4	30
21	10	1200	20	30
22	50	600	5	30
23	55	330	7	60
24	40	3000	6	20
25	100	1800	10	10
26	120	120	12	30
27	200	90	5	15
28	24	1200	5	25
29	22	1200	8	25
30	10	300	6	15
Average	49.76	878.8	6.03	25.5

implement the games. For this reason, since experts' numbers are not real, but hypothetical, we describe a rough comparison.

6.2.1 RQ2.1 - Learning Time

Without having any knowledge in the development of classic multiplayer 2D board games for Android, developers would need to firstly learn many concepts, such as game development on Android and 2D graphics. However, in the case of MAndroid, users do not need to learn any of this, but they only need to be able to model, i.e., to define models conforming to the provided metamodels. This means MAndroid users only need to have modeling and board games knowledge.

The time for teaching different MAndroid concepts to the participants at the beginning of the workshop was 90 minutes. Additionally, before the workshop the participants were provided with four video tutorials. The times spent watching these videos are presented in the second column of Table 9. Regarding the time experts think they would need to spend learning all concepts and acquaring all knowledge needed for implementing the game in their development environment of expertise, these are shown in the second column of Table 10.

We see that the average time to learn all concepts for working with MAndroid is approximately 107 minutes, plus 90 minutes allocated to explanations in the workshop. According to experts' opinions, this time should be around 2985 minutes when using other development frameworks. This seems to indicate a likely significant time saving, although real experiments should be undertaken with experts to confirm this. The time for watching video tutorials has a negative very weak correlation with the number of errors found (-0.1) and weak negative correlation with the time for solving each error (-0.2). This means that as the participants spent more time watching the provided materials, they faced less errors in the development with MAndroid. Also, as they became more familiar with this modeling platform, they spent less time solving each error.

6.2.2 RQ2.2 - Development Time

In order to determine which behavioral and structural parts were and were not modeled by the workshop participants, we divided both parts in different features. The structural part includes modeling of (i) board, (ii) tiles, (iii) elements, (iv) players and (v) visual features of game page. The behavioral part consists of (i) action modeling, (ii) appropriate definition of variables and features, (iii) condition modeling, (iv) event modeling, (v) creation of states and the relation between them and (vi) definition of sequences and their repetition. Figures 19 and 20 present the percentage of structural and behavioral features that were modeled by workshop participants, respectively. We can see in Figure 19 that the board is the structural feature more participants modeled, while the visual features of the game was the feature less participants managed to model. Similarly, Figure 20 shows that creation and repetition of sequences is the behavioral feature more participants modeled, while conditions were the feature less participants managed to model.

In any case, the provided metamodels in this research are complete and support all the behavioral and structural parts of classic multiplayer 2D board games, as some of the participants managed to model Tic-Tac-Toe completely. Additionally, we comprehensively modeled three different board games applying the proposed metamodels. However, we believe that it is possible to improve understandability of the participants by providing more comprehensive materials about the game, modeling and framework before and during the workshop. This, together with an analysis on why participants did generally not model certain parts, are subjects of future work.

The development time of the Tic-Tac-Toe game of each workshop participant is shown in the third column of Table 9. Regarding the time experts think they would need to develop the game in their development environment of expertise, they are shown in the third column of Table 10. It was assumed that they had access to all additional resources, such as images or sound, so that they would not need to search for them.

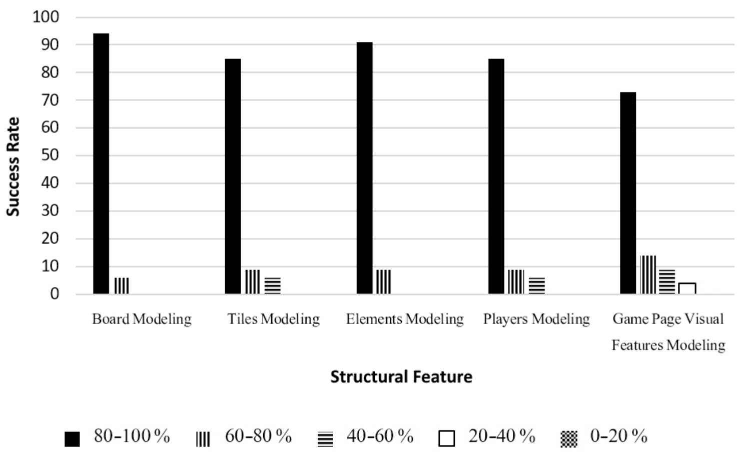


Fig. 19: The percentage of Structural features modeled in the workshop

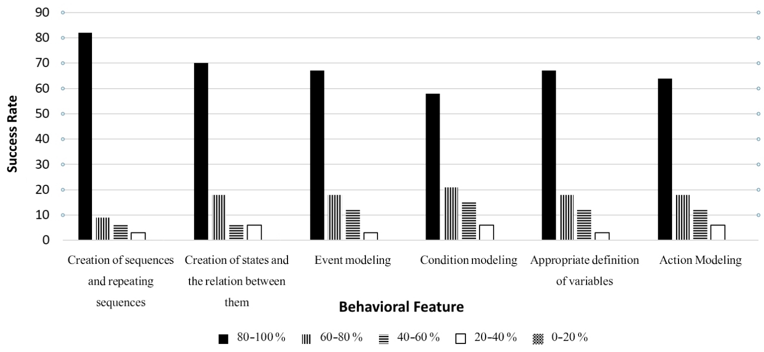


Fig. 20: The percentage of Behavioural features modeled in the workshop

All in all, workshop participants managed to model 89.5% of the game requirements (cf. Figures 19 and 20) in approximately 104 minutes. Therefore, we can estimate that approximately 116 minutes are required to model 100% of the game using MAndroid. On the other hand, according to the opinion of experts, developing the Tic-Tac-Toe game using other frameworks would take around 878 minutes. These results indicate a likely significant saving in development time.

Very weak correlations are presented between game development time and level of knowledge of workshop participants in software modeling, Android, 2D game development, EMF and Ecore (the relations are between 0.05 and 0.10). This means that the level of knowledge of participants does not have a significant influence on game development time with MAndroid, which is a point in favor of our approach. Additionally, very weak correlations are reported between development time and questions 15 to 18 (cf. Section 6.1), which refer to the extensibility of MAndroid. We can conclude that even those participants who spent more time developing the game using MAndroid believe in the extensibility of MAndroid.

6.2.3 RQ2.3 - Time for Solving Errors

After participants finished the modeling phase, the models were checked and syntax errors were fixed and ignored. Then, the Android code was generated automatically in the MAndroid framework by using different model-to-text transformations. After this, the code was run on an Android emulator. If the program did not run properly due to semantic issues, the participant was asked to check the problems and solve them. With semantic errors, we refer to those issues that occur when a statement is syntactically valid, but does not do what the modeler intended [57]. In this evaluation we only focus on semantic errors. For each problem encountered, the number of errors and solving time were recorded (fifth and sixth columns in Table 9). The amount of time for solving each error is shown in the last column of the table.

We found a moderate negative correlation between the time for solving each error and the time spent watching the video tutorials. This means that the longer time the participant spent watching the videos, the shorter time was needed for solving the errors. Additionally, there were week or very week correlations between the time for solving each error and the level of knowledge of participants in software modeling, Android, 2D game, EMF and Ecore. Therefore, the level of knowledge of participants did not influence the time for solving the errors using MAndroid.

Regarding the time for solving each error, workshop participants took an average of 14 minutes using MAndroid. As for experts' opinions, they think they would need around 25.5 minutes on average to solve errors when working with other platforms, showing a likely moderate time saving when working with MAndroid.

6.3 RQ3 - Simplicity

In order to measure the simplitiy of the MAndroid approach and framework, we consider five aspects: (i) level of knowledge about software modeling, (ii) level of expertise in Android programming, (iii) level of expertise in 2D game development, (iv) number of questions asked during Tic-Tac-Toe game development, and (v) simplicity of MAndroid according to the workshop participants' opinions.

According to the results of the first question in the questionnaire of Table 4 (cf. Figure 17), only 12% of participants were familiar with MDE regarding software modeling, and no one was very familiar. Regarding the familiarity with EMF and Ecore (questions 4 and 5), only 18% and 15% of participants were familiar, respectively, and no one was very familiar. Therefore, it can be concluded that it is not essential to have a high level of knowledge in the domains of MDE, EMF and Ecore for using MAndroid. Indeed, workshop participants did not have high expertise in these domains, but they managed to generate approximately 89.5% of the game.

The second criterion (familiarity with Android programming) was asked in question 2 of the same questionnaire. Results show (cf. Figure 17) that the expertise in Android programming of around 71% of participants in the workshop was low or very low. Therefore, it is not essential to have high expertise in Android programming for using MAndroid. Third criterion reaches a similar conclusion: it is not needed to have a high expertise in 2D game development to use MAndroid. Indeed, the responses to question 3 show that only 15% of participants had expertise. However, despite these conclusions, we observed a strong and moderate positive correlations between the answers to questions 3 and 5 and question 10. These relations indicate that the higher the knowledge of participants in EMF and 2D game development, the higher the simplicity of the framework for them.

The number of questions performed by each workshop participant during game development (fourth criterion) is shown in the fourth column of Table 9. Each participant asked an average of 2.5 questions. There is a negative correlation between the number of questions asked and the time spent watching video tutorials, which means that the longer time spent watching the provided videos, the less questions they needed to ask. The number of questions is compared with the questions experts think they would need to ask when implementing the game using other platforms, which are six. These conclusions are complemented with participants' opinions regarding simplicity of using MAndroid. This was asked in question number 10 of questionnaire in Table 4 (responses shown in Figure 17). Responses indicate that 68% of participants strongly believe (27%) or believe (41%) in the simplicity of MAndroid for developing this particular game. There are moderate and strong positive correlations between the answers of participants to question number 10 and questions number 11 to 18. This means that, according to participants' opinions, the higher the simplicity of MAndroid, the more attractive, effective in detecting errors and extensible the MAndroid framework is. Additionally, very weak and weak correlations are reported between question 10 and level of knowledge of participants. This suggests that reasonable responses are received from the workshop participants, which are in turn independent from their level of knowledge.

To compare the simplicity criterion of MAndroid with other approaches, different questions were asked to the experts (cf. Table 7). The responses in Figure 18 indicate that approximately 67% of experts strongly believed (20%) or believed (47%) that it is required to have a high level of knowledge in Android programming. Additionally, 50% strongly believed (17%) or believed (33%) that it is important to be expert in 2D game development. However, only 3% thought is is important to have expertise in software modeling.

Overall, it can be concluded that for game development in MAndroid it is not essential to have a high level of knowledge in Android development and 2D game development. However, these aspects are important for developing this kind of games using other platforms. Furthermore, using MAndroid we are likely to face less issues, since the number of questions workshop participants needed to ask was smaller than the number of questions experts think they would have faced.

6.4 RQ4 - Attractiveness

We designed questions 11, 13 and 14 shown in Table 4 to measure how attractive MAndroid was for workshop participants. Results in Figure 17 show that more than half of the participants found MDE attractive (35%) or very attractive (29%)—question 13—, and half of them found game development attractive (38%) or very attractive (15%)—question 14. Additionally, 56% of participants thought MAndroid was very attractive (12%) or attractive (44%)—question 11. Strong and moderate correlations between questions 11 to 14 and questions 6 to 9 are observed. They indicate that the higher participants think the applicability of MAndroid is, the more attractive they find the approach. Similar strong positive correlations are found between answers to questions 11 to 14 and questions 10 and 12. These suggest that the more the participants believe in simplicity and effectiveness of MAndroid in finding errors, the more attractive MAndroid for them is. Finally, we found strong correlations between questions 11 to 14 and questions 15 to 18. This means that the higher the attractiveness of MAndroid for the participants, the stronger they believe in its extensbility.

6.5 RQ5 - Technical Evaluation

In order to answer this RQ, we compare the weight of APK files, memory occupation and CPU usage of the final Tic-Tac-Toe game generated with MAndroid with respect to seven state-of-the-art implementations of the game.

The compared games are similar for general features of the Tic-Tac-Toe game and style, but have many differences in terms of technical features. Therefore, the provided comparison is limited to the mentioned technical features and not focused on any other detailed features. Considering all features in depth is out of scope in this research. Furthermore, the games with which we have compared our generated game might have more functionality other than the functionality available in our generated game, which is a threat to the validity of the following results.

The results are presented in Table 11, where the last row displays the information of the game generated with MAndroid. We can see that the technical details of the game obtained with MAndroid are satisfactory. First, it is the game that weights the least, and the CPU usage is also very positive. The worst feature is the memory occupation (8.1%), although it is not too bad.

Table 11: Android technical evaluation for Tic-Tac-Toe game

Developer	**Volume (MB)**	**Memory Usage**	**CPU Usage**
Wintrino [49]	18.45	4.9%	8.7%
Arclite Systems [50]	50.08	5.9%	28.4 %
BYRIL [51]	29.17	6.2%	30.0%
Fun Games Free [52]	42.24	5.0%	55.0%
Big Brain Kraken [53]	13.25	7.6%	11.0%
Plauzio [54]	22.11	3.4%	38.6%
AlmaTime [58]	14.56	4.5%	46.6%
MAndroid	3.16	8.1%	6.6%

6.6 Discussion

A summary of the evaluation results is shown in Table 12. In terms of applicability, it has been demonstrated that it is possible to generate different classic multiplayer 2D board games without the need of writing any piece of code. Furthermore, it should be extensible for other multiplayer 2D games and for different mobile platforms. Regarding developer performance, the time for the different stages related to the game development are likely reduced. First, the time for learning the necessary concepts for developing the game. Second, the time for developing the game, which in case of MAndroid refers to time for modeling the game. Third, the time for discovering and solving semantic errors.

The approach offers a good level of simplicity, since it is not necessary to have expertise in Android game development. This is because all configuration files as well as the final code are automatically generated with a set of model-to-text transformations. The MAndroid framework appears to be attractive for the users, and they have managed to model a high percentage of the Tic-Tac-Toe game structure and behavior. A study and analysis on the parts that most participants did not manage to model is out of scope of this paper. However, this is an interesting point and is subject of future study. Regarding the technical evaluation, the Tic-Tac-Toe game generated with MAndroid manages to reduce the volume of APK files generated as well as the CPU usage, while it occupies more memory. This is in comparison with other seven state-of-the-art implementations of the game. Finally, despite energy efficiency is out of scope of this paper, it is still interesting to discuss it. In mobile applications, energy usage depends on the usage of CPU, memory, sensors and network. The games we consider do not employ sensors and network, so energy consumption depends on CPU and memory usage. While the game generated with MAndroid uses less CPU than the other games, it uses more memory. For this reason, we cannot predict with confidence improvements on energy efficiency, so a specific study is needed.

In summary, from the provided evaluation, we can conclude that the MAndroid approach is applicable for the development of classic multiplayer 2D board games, and it reduces the developer performance time. Additionally, it

Table 12: Summary of MAndroid evaluation

Research Question	Result of MAndroid evaluation
RQ1 - Applicability	Possibility for development of different kinds of classic multiplayer 2D board Games with no need for writing code Extensible for other classic multiplayer 2D games and for other mobile platforms (according to 88% participants)
RQ2 - Developer performance	Time for learning concepts related to game development is likely significantly reduced Game development time is likely significantly reduced Time for discovering and solving errors is likely reduced
RQ3 - Simplicity	It is not essential to have high expertise in game and Android development (71% participants did not have it)
RQ4 - Attractiveness	It is an attractive approach and enhances the interest of users for this domain (according to 56% participants)
RQ5 - Technical Evaluation	Reduces the volume of APK files and CPU usage

is simple, extensible and attractive. In terms of technical evaluation, it reduces size of APK files and CPU usage.

However, despite being satisfied with the results, our approach is targeted at games with specific features for the Android OS. Increasing the amount of game features to consider in MAndroid is a challenge, and we hope to include more features in the upcoming versions of our approach and tool.

6.7 Threats to Validity

In the following we describe the four types of threats that can affect the validity of our study, according to Wohlin et al [59].

6.7.1 Construct validity threats

They are concerned with the relationship between theory and what is observed. The comparison of the proposed framework with others is conducted with different questionnaire forms. Therefore, a possible construct validity threat is the fact that we have not considered the same users to evaluate MAndroid with other approaches. However, it was not trivial to find appropriate participants to accept participating in this type of evaluation. We plan to perform a more detailed evaluation of MAndroid with other approaches as future work.

6.7.2 Conclusion validity threats

Threats to the conclusion validity are concerned with the issues that affect the ability to draw correct conclusions from the data obtained from the experiments. In our experiments, some of the participants did not complete the workshop and did not continue with solving their errors. This may threat the calculation of average time for solving errors in the workshop. To avoid this threat we omitted those participants when measuring the average time for solving errors.

6.7.3 Internal validity threats

Instrumental bias. It concerns the consistency of measures over the course of the experiment. To ensure consistency, all the participants of the workshop developed the Tic-Tac-Toe game according to the predefined specification. Additionally, all measures taken during the workshop were recorded in the same manner in order to ensure consistency.

Selection bias. Participants of the workshop were selected from different graduate, master and PhD students. This means there might be a bias in our participants with respect to the level of knowledge in MDE, Android and game development.

6.7.4 External validity threats

These threats have to do with the extent to which it is possible to generalize the findings of the experiments. The first threat is that the framework is evaluated with only three board games, which externally threatens the generalizability of our results. To mitigate this threat, the authors selected three classic board games, such that all the structural and behavioral features of classic multiplayer 2D board games are covered at least in one of these three games (cf. Tables 2 and 3). We do believe that the provided metamodels in this research are complete and generalizable to any classic board game and the presented framework enables to generate 100% correct code for any board game. Another threat is related to the fact that the technical comparison with other developed games does not take into account extra features provided by such games.

7 Conclusion and future work

This paper presents MAndroid, an approach and framework that applies an MDE methodology for the development of classic multiplayer 2D board games in Android. With different metamodels integrated in the framework, the user does not need to implement the game, but to model it. Then, a predefined set of model-to-text transformations deal with the generation of the final Android code. This means that users do not need to have any Android knowledge. Three different games have been implemented, demonstrating the applicability of the approach. Furthermore, a workshop was organized in which different users evaluated different dimensions of MAndroid. Its simplicity, extensibility and attractiveness were positively evaluated. In particular, surveyed participants believe the tool should be extensible for other multiplayer 2D games and for different mobile platforms. Also, results suggest that development time is reduced by using MAndroid, and it appears to be attractive for the users. Regarding some technical aspects such as memory and CPU usage, the game generated with MAndroid seems to consume less CPU than other implementations in the market, although it uses more memory.

There are some lines of future work that we would like to address next. First of all, we want to study and analyse why most participants were not able to model certain parts of the Tic-Tac-Toe game. Also, we would like to extend the proposed framework for multiplayer games where players play on different devices. Since we are following an MDE methodology, it is possible to generate code for other platforms different than Android. For this, we would need to develop new model-to-text transformations. We want to explore the possibility to generate code for iOS devices. We would like to consider more types of games in our framework, too. Finally, we plan to improve the graphical editor environment, so that the framework usability is enhanced.

Verifiability

For the sake of verifiability, our MAndroid framework as well as the implementation of the three games (Tic-Tac-Toe, Othello and Backgammon) are available on our GitLab website [45].

References

1. E. E. Thu and N. Nwe, "Model driven development of mobile applications using drools knowledge-based rule," in *2017 IEEE 15th International Conference on Software Engineering Research, Management and Applications (SERA)*. IEEE, 2017, pp. 179–185.
2. E. R. N. Valdez, Ó. S. Martínez, B. C. P. G. Bustelo, J. M. C. Lovelle, and G. I. Hernandez, "Gade4all: developing multi-platform videogames based on domain specific languages and model driven engineering," *IJIMAI*, vol. 2, no. 2, pp. 33–42, 2013.
3. A. Rollings and E. Adams, *Andrew Rollings and Ernest Adams on game design.* New Riders, 2003.
4. J. Novak, *Game development essentials: an introduction.* Cengage Learning, 2011.
5. M. Brambilla, J. Cabot, and M. Wimmer, "Model-driven software engineering in practice," *Synthesis lectures on software engineering*, vol. 3, no. 1, pp. 1–207, 2017.
6. F. E. Hernandez and F. R. Ortega, "Eberos gml2d: a graphical domain-specific language for modeling 2d video games," in *Proceedings of the 10th Workshop on Domain-Specific Modeling.* Citeseer, 2010, pp. 1–1.
7. "Android developers," http://developer.android.com/index.html/, 2019.
8. "Smartphone market share," http://www.idc.com/prodserv/smartphone-os-market-share.jsp/, 2019.
9. J. P. Hinebaugh, *A board game education.* R&L Education, 2009.
10. F. Budinsky, D. Steinberg, R. Ellersick, T. J. Grose, and E. Merks, *Eclipse modeling framework: a developer's guide.* Addison-Wesley Professional, 2004.
11. "Eclipse acceleo project," https://www.eclipse.org/acceleo/, 2019.
12. Merriam-Webster, "Definition of game by merriam-webster," http://www.merriam-webster.com/dictionary/game/, 2019.
13. F. T. Tschang, "Videogames as interactive experiential products and their manner of development," *International Journal of Innovation Management*, vol. 9, no. 01, pp. 103–131, 2005.

14. D. Callele, E. Neufeld, and K. Schneider, “Requirements engineering and the creative process in the video game industry,” in *13th IEEE International Conference on Requirements Engineering (RE’05)*. IEEE, 2005, pp. 240–250.
15. E. R. Núñez-Valdez, V. García-Díaz, J. M. C. Lovelle, Y. S. Achaerandio, and R. González-Crespo, “A model-driven approach to generate and deploy videogames on multiple platforms,” *Journal of Ambient Intelligence and Humanized Computing*, vol. 8, no. 3, pp. 435–447, 2017.
16. K. Sánchez, K. Garcés, and R. Casallas, “A dsl for rapid prototyping of cross-platform tower defense games,” in *2015 10th Computing Colombian Conference (10CCC)*. IEEE, 2015, pp. 93–99.
17. C. Kelly, *Programming 2D games*. AK Peters/CRC Press, 2012.
18. A. Developer, “Android developer,” *línea]. Available: https://developer. android. com*, 2015.
19. R. Meier and I. Lake, *Professional Android*. John Wiley & Sons, 2018.
20. U. Frank, “Domain-specific modeling languages: requirements analysis and design guidelines,” in *Domain Engineering*. Springer, 2013, pp. 133–157.
21. M. Gharaat, M. Sharbaf, B. Zamani, and A. Hamou-Lhadj, “Alba: A model-driven framework for the automatic generation of android location-based apps,” *Automated Software Engineering*.
22. A. Kleppe, *Software language engineering: creating domain-specific languages using metamodels*. Pearson Education, 2008.
23. D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.
24. S. Sendall and W. Kozaczynski, “Model transformation: The heart and soul of model-driven software development,” *IEEE software*, vol. 20, no. 5, pp. 42–45, 2003.
25. J. Bézivin, “In search of a basic principle for model driven engineering,” *Novatica Journal, Special Issue*, vol. 5, no. 2, pp. 21–24, 2004.
26. J. Oldevik, T. Neple, R. Grønmo, J. Aagedal, and A.-J. Berre, “Toward standardised model to text transformations,” in *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 2005, pp. 239–253.
27. J. Musset, É. Juliot, S. Lacrampe, W. Piers, C. Brun, L. Goubet, Y. Lussaud, and F. Allilaire, “Acceleo user guide,” *See also http://acceleo. org/doc/obeo/en/acceleo-2.6-user-guide. pdf*, vol. 2, p. 157, 2006.
28. H. Tufail, F. Azam, M. W. Anwar, and I. Qasim, “Model-driven development of mobile applications: A systematic literature review,” in *2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. IEEE, 2018, pp. 1165–1171.
29. M. Usman, M. Z. Iqbal, and M. U. Khan, “A model-driven approach to generate mobile applications for multiple platforms,” in *2014 21st Asia-Pacific Software Engineering Conference*, vol. 1. IEEE, 2014, pp. 111–118.
30. H. Heitkötter, T. A. Majchrzak, and H. Kuchen, “Cross-platform model-driven development of mobile applications with md2,” in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, 2013, pp. 526–533.
31. A. G. Parada and L. B. De Brisolara, “A model driven approach for android applications development,” in *2012 Brazilian Symposium on Computing System Engineering*. IEEE, 2012, pp. 192–197.
32. A. G. Parada, E. Siegert, and L. B. De Brisolara, “Generating java code from uml class and sequence diagrams,” in *2011 Brazilian Symposium on Computing System Engineering*. IEEE, 2011, pp. 99–101.
33. M. Kosanović, I. Dejanović, and G. Milosavljević, “Applang–a dsl for specification of mobile applications for android platform based on textx,” in *AIP Conference Proceedings*, vol. 1738, no. 1. AIP Publishing LLC, 2016, p. 240003.
34. M. Ko, Y.-J. Seo, B.-K. Min, S. Kuk, and H. S. Kim, “Extending uml meta-model for android application,” in *2012 IEEE/ACIS 11th International Conference on Computer and Information Science*. IEEE, 2012, pp. 669–674.
35. A. Sabraoui, M. El Koutbi, and I. Khriss, “Gui code generation for android applications using a mda approach,” in *2012 IEEE International Conference on Complex Systems (ICCS)*. IEEE, 2012, pp. 1–6.

36. A. Sabraoui, A. Abouzahra, K. Afdel, and M. Machkour, "Mdd approach for mobile applications based on dsl," in *2019 International Conference of Computer Science and Renewable Energies (ICCSRE).* IEEE, 2019, pp. 1–6.

37. M. Lachgar and A. Abdali, "Rapid mobile development: Build rich, sensor-based applications using a mda approach," *IJCSNS*, vol. 17, no. 4, p. 274, 2017.

38. C. Rieger and H. Kuchen, "A process-oriented modeling approach for graphical development of mobile business apps," *Computer Languages, Systems & Structures*, vol. 53, pp. 43–58, 2018.

39. S. Vaupel, G. Taentzer, R. Gerlach, and M. Guckert, "Model-driven development of mobile applications for android and ios supporting role-based app variability," *Software & Systems Modeling*, vol. 17, no. 1, pp. 35–63, 2018.

40. M. Núñez, D. Bonhaure, M. González, and L. Cernuzzi, "A model-driven approach for the development of native mobile applications focusing on the data layer," *Journal of Systems and Software*, vol. 161, p. 110489, 2020.

41. V. Guana and E. Stroulia, "Phydsl: A code-generation environment for 2d physics-based games," in *2014 IEEE Games, Entertainment, and Media Conference (IEEE GEM)*, 2014.

42. E. Marques, V. Balegas, B. F. Barroca, A. Barisic, and V. Amaral, "The rpg dsl: a case study of language engineering using mdd for generating rpg games for mobile phones," in *Proceedings of the 2012 workshop on Domain-specific modeling*, 2012, pp. 13–18.

43. D. Altunbay, E. Cetinkaya, and M. Metin, "Model driven development of board games," in *the First Turkish Symposium on Model-Driven Software Development (TMODELS)*, 2009.

44. E. M. Reyno and J. Á. Carsí Cubel, "Automatic prototyping in model-driven game development," *Computers in Entertainment (CIE)*, vol. 7, no. 2, pp. 1–9, 2009.

45. M. Derakhsandi, "MAndroid Framework," https://gitlab.com/mohammad71/mandroid/, 2019.

46. "Gametable," https://gametable.org.

47. "Cardgames," https://cardgames.io/reversi/.

48. G. Albaum, "The likert scale revisited," *Market Research Society. Journal.*, vol. 39, no. 2, pp. 1–21, 1997.

49. "wintrino," https://play.google.com/store/apps/details?id=com.tictactoe.wintrino/.

50. "Arcsys," https://play.google.com/store/apps/details?id=com.arcsys.tictactoe.lite.free.puzzle.games/.

51. "byril," https://play.google.com/store/apps/details?id=com.byril.tictactoe2/.

52. "Fun games free," https://play.google.com/store/apps/details?id=tic.tac.toe.games.board.kids.tictactoe.free/.

53. "Big brain kraken," https://play.google.com.bigbrainkraken.tictactoe/.

54. "Playzio," https://play.google.com.playzio.tictactoefree/.

55. P. Schober, C. Boer, and L. A. Schwarte, "Correlation coefficients: appropriate use and interpretation," *Anesthesia & Analgesia*, vol. 126, no. 5, pp. 1763–1768, 2018.

56. H. Akoglu, "User's guide to correlation coefficients," *Turkish journal of emergency medicine*, vol. 18, no. 3, pp. 91–93, 2018.

57. R. S. Pressman, *Software engineering: a practitioner's approach.* Palgrave macmillan, 2005.

58. "almatime," https://play.google.com/store/apps/details?id=com.almatime.tictactoe/.

59. C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering.* Springer Science & Business Media, 2012.

A Appendix

This Appendix presents the full version of the metamodels and describes them in detail.

A.1 Game Menu Metamodel

This section extends the explanations presented in Section 4.1.1. Figure 21 displays the complete menu metamodel. The concepts that appear in the metamodel are presented next.

***Game*:** It is the root of the metamodel, from which we start to build the game menu.

***Designer*:** Every game normally keeps essential information on the game designer in this component. Such information includes the designer's name and contact information. Although it is possible to add the name and details of the game designer as attributes in the *Game* class, we emphasize its role creating the *Designer* class. Besides, this is more convenient if we consider future extensions of our tool, such as the possibility of having several designers.

***Menu*:** The game menu is composed of a combination of pages in our MAndroid approach. A page can either be a *MenuPage* or a page where the main process of the game is started.

***MenuPage*:** Menu pages are like a canvas, so we can design their user interface by using different graphical elements. Here, the user can build as many pages as desired. The order of these pages is considered.

***SplashPage*:** Most mobile phone games contain a page for some general information such as name, version and developer. This page opens after entering the game environment and, after a few seconds, it transfers the game execution to the next page. Such a page can be placed in the game using the *SplashPage* component.

***GamePage*:** There is always a page in the game for the main process. The main game process in a board game takes place by throwing a die, or moving an element. This page is built with the help of the *GamePage* class. For instance, Figure 4 shows all the pages of the Tic-Tac-Toe game developed using MAndroid. As shown in the figure, the game is built using three pages, including, from left to right, a *SplashPage*, a *MenuPage* and a *GamePage*.

***Variable*:** *Variable*s are used to give the user the ability to change desirable components placed in the game, what allows to use not only predefined settings. A *Variable* needs a name, a type and a desired value that could be later modified.

***Resource*:** This component is used to add resources like images, sounds and XML files to the game. All these resources must be later materialized as separate files to be used to build the game menu. The file names are used to identify them. It is important to consider the Android OS rules when naming these files as it does not allow using repetitive names and capital letters. As future work, we plan to let the user specify any name for these files and automatically convert these names to Android valid names.

***MainFrame*:** Every menu page consists of a main frame. As its name suggests, it acts as a frame that holds all other components of the game menu, such as control components. This frame can be horizontal or vertical depending on the game implementation patterns.

***Box*:** MAndroid introduces a method called the *Box Design Method* to design the user interface of every page. This enables the user to design the user interface of every page without being involved in the technical details related to Android user interface designing such as layouts. Most of the designs that are built using *LinearLayout*, *GridLayout* and *TableLayout* in Android will be implementable using this method. This method divides every page into a set of columns and rows, which are placed together to build a game page. Then, a concept named *Box* is used to place various graphic components in the page. Every box is in fact a space in the page that can hold a graphic component. Adjusting the size of these boxes on each page is done in proportion to the display size, so that the design can be properly displayed on devices of different sizes and responsive designs can be created. Also, boxes can contain other boxes with our approach to create more complex designs. Figure 5 shows how a *MenuPage* is displayed using this method. The page is made up of 5 different rows and each row holds a number of boxes as well. For example, the second row from the top holds three boxes. The boxes on the right and left are empty spaces that have been

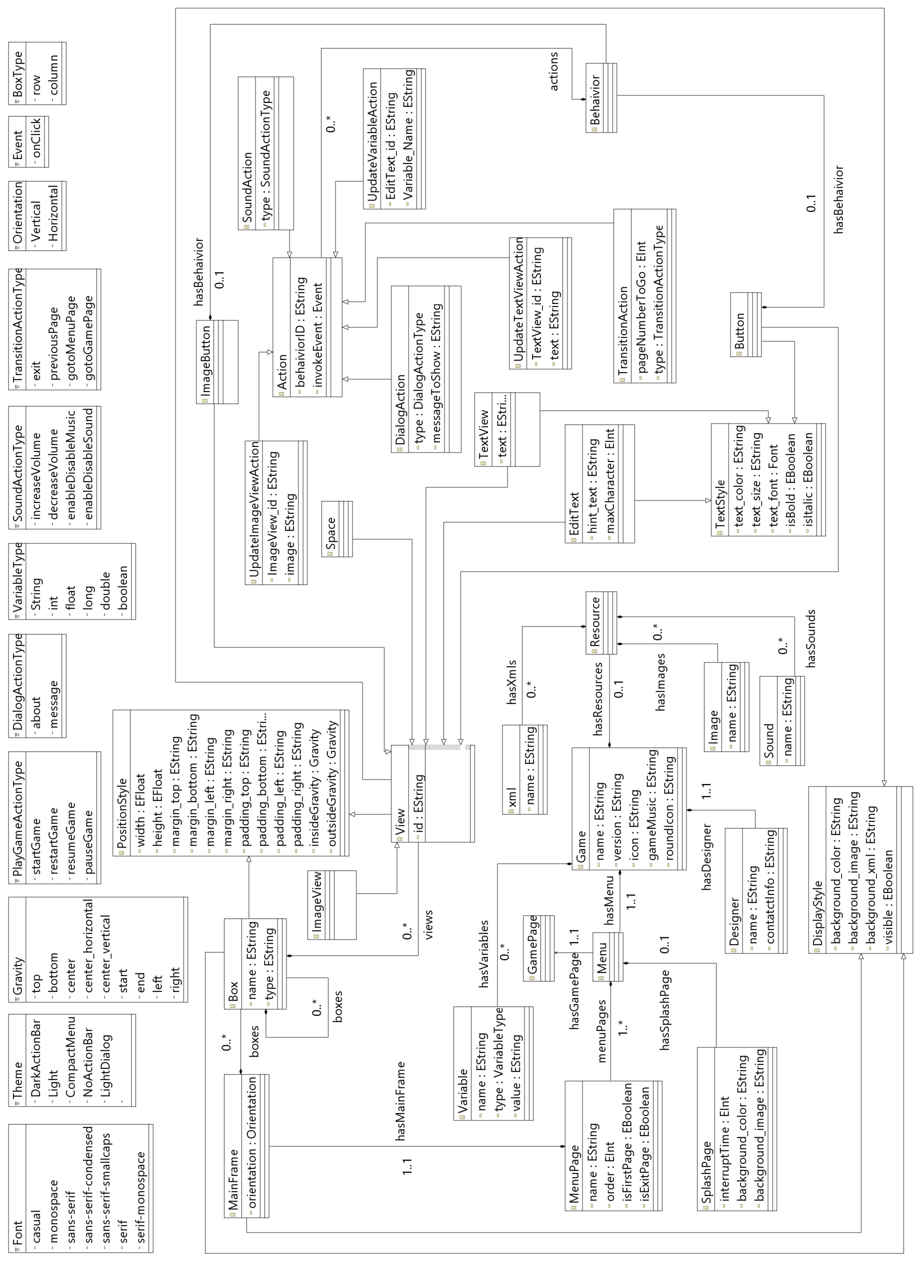


Fig. 21: The Menu Metamodel for generation of 2D board game in MAndroid

placed on both sides to align the row. The box in the middle has been placed as a space to hold a button.

***View*:** This component is used to display the Android user interface components in a page. Every view needs to have a unique identity in the model, which is used to identify it in the entire model. To enhance the understandability of users, the real names of Android components are used in the menu metamodel.

According to the concepts mentioned earlier, it is possible to define the relationship between the game's menu metamodel and the *Game*, *Menu*, *MenuPage*, *MainFrame*, *Box* and *View* components so as to build the user interface. This approach makes it compulsory for every game to have a *Menu* component that may contain one or more *MenuPage*s. Every *MenuPage* needs to have a main frame, represented by the *MainFrame* component. It is possible to place as many *Box*es as we want inside the frame. Additionally, every box can contain a number of boxes or *View* type components. This provides a convenient way for the user to generate the user interface for each page.

Since the game menu does not require too many *View* components, we have used and defined only a selected number of Android *Views* in this metamodel, including *Button* for creation of buttons, *ImageButton* for creation of images as a button, *ImageView* for showing images, *TextView* for showing text, *EditText* for getting input data from the user and *Space* to create blank areas inside a box.

Then, to present visual features in the metamodel, we have defined three classes, namely *DisplayStyle*, *TextStyle* and *PositionStyle*. These classes are separated, as some Views may not need all styles. The remaining metamodel components are defined next.

***DisplayStyle*:** The display properties of every part of the user interface page can be defined using the *DisplayStyle* component. Examples of these characteristics are the background image or color.

***PositionStyle*:** This component is used to organize and align all properties related to placing the boxes or other graphic coordinates like width, height, and margins, among others, in the page.

***TextStyle*:** In order to define properties like color, font or size, which are related to displaying text in components such as *EditText* or *TextView* that use text, the *TextStyle* component is used.

***Behavior*:** It is possible to assign behavioral properties to graphical components in the game menu metamodel so that the considered action is performed on user interaction, such as pressing a key or a button. Executable behaviors are defined under the *Action* component.

***Action*:** This component is used by a graphic component after an event is called in order to define the needed behavior. For example, when a key on the menu is pressed, the "clicking" event is called and naturally the behavior defined to respond to this event is accrued. In MAndroid, only very common actions in games have been defined. For instance, *UpdateImageViewAction* is used to change the image of a display component. *SoundAction* allows to perform audio actions, such as reducing or increasing the volume and playing/pausing music, while *DialogAction* is used to display an "about" window or a message in the game menu. Regarding updating actions, *UpdateVariableAction* is used to update and refresh the value of the variables defined in the game menu, for which it needs to receive the corresponding updated value from the user using an *EditText* component, while *UpdateTextViewAction* may be used to change the text displayed by a *TextView* component in the game page. Finally, since this approach is based on a set of pages, one of the most important requirements is the possibility for shifting between pages. This can be a shift between the current page and the next or previous ones, or to a specific page number, to the *GamePage* or exiting the game. This is managed by the *TransitionAction* component.

A.2 Game Structure and Behavior Metamodel

This section extends the explanations given in Section 4.1.2. Figure 22 displays the complete metamodel proposed for the game behavior and structure according to the MAndroid approach. In the following, explanations for the different components of the metamodel are provided.

***Game*:** Similar to the game menu metamodel, modeling is started from the most basic component, which is the game component. This component provides integration between the menu and game metamodels to have a uniform understanding between them.

***GamePlayFeatures*:** Some general features of the game, such as the number of players, use of dice and the type of dice in the game are defined with the help of this component.

***UIFeatures*:** As mentioned earlier in the game menu metamodel, the declaration of a user interface related to the process of a board game is performed in the *GamePage* page. Because of the differences between the implementation techniques in static and dynamic user interfaces, all the graphic coordinates in the *GamePage* have to be drawn completely independent from other pages and in a dynamic way using the graphic libraries of Android. However, the user interface in the game menu pages are of a static form based on XML files. To be able to create diverse *GamePage* pages, some features of this page are abstracted and are placed in the *UIFeatures* component. This enables the user to personalize the game page display by allocating desired quantities to these features.
For instance, Figure 7 represents the general pattern designed for the *GamePage* in MAndroid. The design is made up of three parts or panels. *Top Panel* contains the menu button and two spaces for displaying information of the two players. These spaces are provided depending on the number of players playing the game, and are called *Player Box*. The middle panel, declared as the *Board Panel*, contains the required space to place the board component. The *Bottom Panel* can contain the throw dice button and two other spaces to display the information of two additional players, depending on the number of players and game style. Every *Player Box* contains two different spaces to accommodate desired texts, which might be the player's name, points, turns, etc. This space is separated by a line that passes through the box. Every *Player Box* is made up of an internal and an external space, which may have different backgrounds.

***GameVariable*:** Undoubtedly, designing a game requires declaration of many variables. Although many of these variables are declared in our MAndroid approach as prerequisites, the *GameVariables* component allows to declare further variables.

***Board*:** The most important and basic concept in the structure of a board game is the board. In board games, the board is the place that accommodates all the components of the game. Naturally, every board game needs to have one board and the game cannot be executed without it.

***Tile*:** In the MAndroid approach, every board is made up of several matrix components called tiles. These tiles are locations in which game elements can be allocated. Since the board has to be a rectangular or square space, every tile can be easily identified using its row and column index number. The numbering of rows and columns is started from the top left corner with zero. The attribute number is used to calculate the row and column of tile, which makes it more simpler for the user and also it is more dynamic in the process.

***Player*:** Since MAndroid aims to generate multiplayer games, we need the concept of player in our metamodel. It is not possible to declare more than four players in order to consider only those games that can be displayed on a single screen. Additionally, by defining more than four players, the displaying limitations for presenting all the details of the game may happen.

***Element*:** The element is an important component in board games. The progress of a board game is highly dependent on the movement or position of elements. Every element belongs to only one player and must always be in a tile. In every board game, players may have multiple and different elements; for instance, in chess, a player may have elements like the pawn, knight, bishop, etc. Generally, the structure of board games in MAndroid is formed by the board, tiles and the elements. Figure 11 displays this concept in the Othello game. In the figure, the board is made up of 8 columns, 8 rows and therefore 64 tiles.MAndroid supports both structural and behavioral features. It is possible to assign any image to an element, in terms of its structural aspect. Additionally, it is possible to modify the behavior of the element. Thus, elements move between tiles, get visible and hide, and are added or removed from the board. These are the required operations for an element in a real board game.

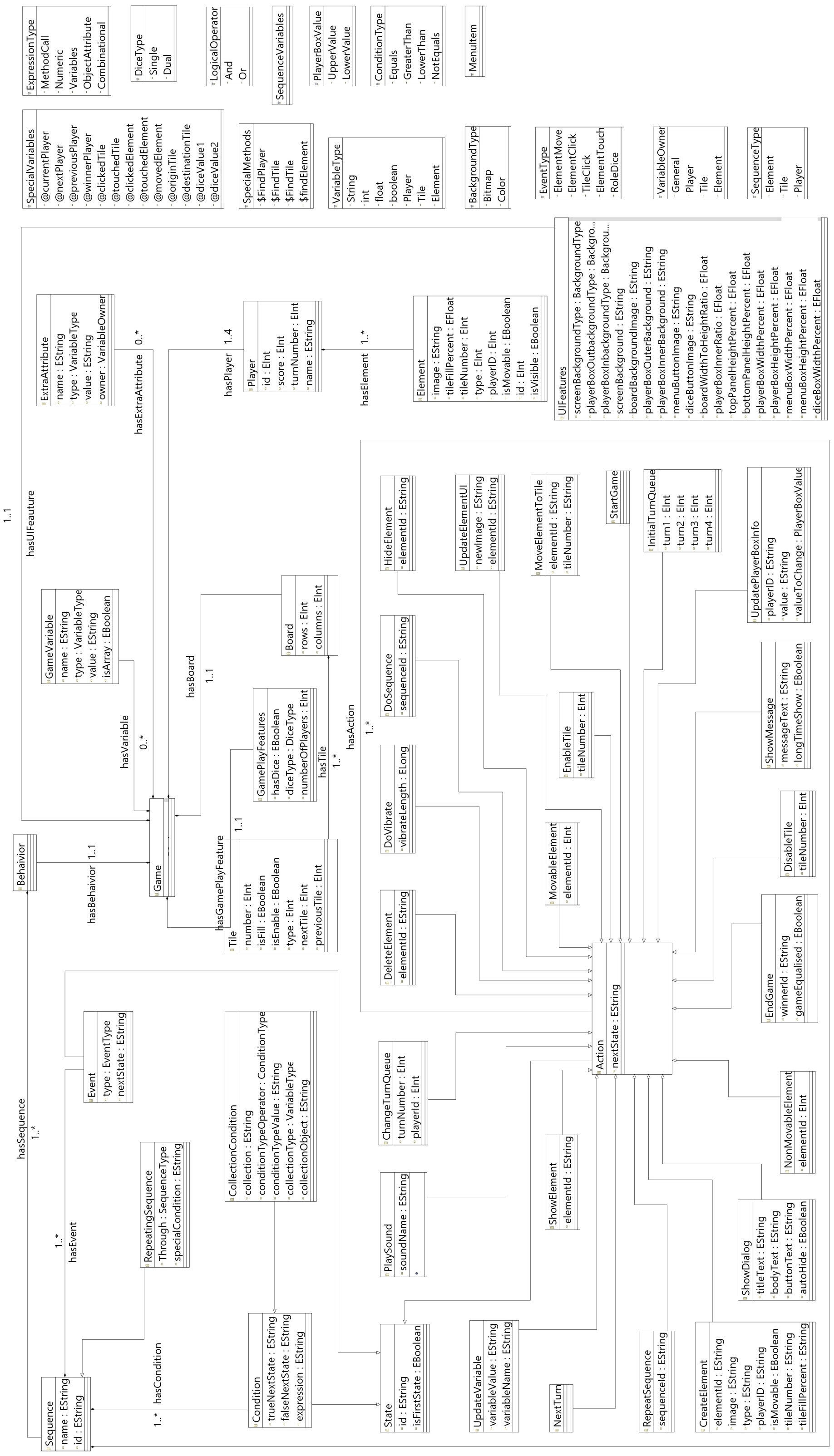


Fig. 22: The game Structure and behavior Metamodel in MAndroid

***Behavior*:** The most challenging part in designing a game is defining its rules. UML structural diagrams such as class diagrams do not provide all necessary features to model the behavior and rules of the game. Therefore it is necessary to use UML behavioral diagrams, such as state machines. For this purpose, a new method has been proposed in the MAndroid framework that combines the concepts used in class diagrams and state machines in order to model the game behavior using Ecore diagrams.

The game behavior in MAndroid is made up of three components including condition, action and event. *Event*s are executed by the player during the game. For example, throwing the dice is an event that is called by pressing on the dice icon. If an event component is reached during the game flow, the flow will be stopped until the event is called. *Condition* is another component of the game used to check the correctness of statements and, in turn, decide on the continuation of the game. For example, after an action by the user, in case the element lands on a tile with even number, the game will continue on path A; and if the element lands on an odd number tile, it will continue on path B. *Action* is another component used to perform several actions that have been defined under the domain of the game. For example, deleting an element from the board when hit by an enemy element is an action. Figure 12 displays the principles of behavioral modeling in MAndroid. According to the figure, the game starts at a singular point and ends in another point. After the game starts, at every point of time, it is at a specific point called *State*, which can be either an event, action or condition component. A concept named *Sequence* has been used in this approach to ease the modeling and prevent the implementation of repetitive states. Every sequence is made up of a set of states and does not create any change in the game by itself; instead, it is used to organize and keep in order the states to be used. In case the need to use a sequence more than once arises, the *RepeatingSequence* component can be used. Figure 12 shows a game that uses 3 sequences and 12 different states. The game enters a state in the form of an event. The flow of the game stops until the intended event is called by the user. Then the game reaches a state or condition where the game can be either directed to the right or left. Finally, if the game is directed towards its ending point, it will end. All behavioral states and rules of the game can be modeled using this method.

***Sequence*:** As mentioned, every sequence is composed of a set of events, conditions and actions. In this approach, the game needs to have one or an unlimited number of sequences to hold the game states. The sequences may be identified and used later for a particular operation with the help of their unique identities. In case there is the need to use a sequence repeatedly, the concept of *RepeatingSequence* can be applied.

***State*:** In this approach, every state is a unique and effective point in the game that helps to guide the game from its starting point to its ending point. The game flow starts from a point where the *isFirstState* attribute is true and then the game keeps moving forward with the help of the event's *nextState* attribute, which stores the unique identification of the next state, until one of the ending points of the game is reached.

***Event*:** Events are elements that are executed by players and change the flow of the game. The flow of the game is stopped until an event is called by the user. In the MAndroid approach, all events that can possibly occur in a board game have been covered. Examples of these events are throwing the dice, moving an element form an origin tile to a destination tile, touching an element, clicking on it or pressing a tile.

***Condition*:** Sometimes it is necessary to guide the flow of a game to different paths based on the execution of certain events and states or upon satisfaction of certain conditions. The *Condition* component is used to accomplish this task in the MAndroid approach. Condition is the only state in the game whose output leads to two different paths, whereas all other states have only one output. The condition component has an *expression* attribute that is a logical statement and can acquire true and false as its value. If the expression is true, the game is guided towards a path whose *trueNextState* attribute is true, otherwise it is guided to the *falseNextState* feature. To write an expression in a condition, one must follow the grammar rules of Java, reason why these three attributes are of type EString. Also, it is possible to use predefined as well as user-defined variables. Another conditional component is the *CollectionCondition*, which is used for writing complex conditions that are applied on a collection of components such as the elements or the tiles.

***Action*:** The action component is used to undertake an action in the game. The actions have been designed by considering the domain of board games, so that all needed actions are possible. All possible actions in the MAndroid approach are explained as follows:

– *DoSequence*: This action is used to start a sequence from any point. This ability may be used to easily search the predefined sequences for a desirable point from where to start the game.
– *RepeatSequence*: This action is used if repetitive sequences are needed. With the help of this action, the desired sequence is executed once again from the start.
– *UpdateVariable*: This action is used in case it is required to change the value of the defined variables.
– *StartGame*: The action can be used to start the game with showing a welcome message to users.
– *ShowDialog* and *ShowMessage*: One of the most important needs of a mobile phone is the display of required messages to the players. These messages can be reminders about the game rules, error displays and displays of players' scores. Dialogs are separate windows that open on the current page and stay on the screen until their window is closed. Messages are also a form of information that stay on the screen for some time and disappear shortly after. Figure 13 shows an example of a Dialog window and a message window in MAndroid.
– *InitialTurnQueue*: Applying turns is an important concept in board games. The process of identifying the turns on the starting point may be different depending on the game. In our approach, it is possible to implement it as desired using the *InitialTurnQueue* component. This action determines the initial turn as the game starts such that all players are placed sequentially in a queue. Every player gets in the end of the queue after their turn.
– *ChangeTurnQueue*: If the need arises to change the turns sequence while playing the game, this action is used. For example, we might want to relocate or completely remove a player form the turn queue. The *ChangeTurnerQueue* is used in such circumstances.
– *NextTurn*: This action is used to shift the turn to the next player in the queue.
– *HideElement*: Every element has a property called *isVisible*, which determines whether the element is visible on the board or not. Invisible elements cannot be seen on the game screen, although they do occupy the position in a tile. To hide an element, the *HideElement* action is used.
– *ShowElement*: It is used to make an invisible element visible on the board, i.e. it is directly opposite to *HideElement*.
– *DeleteElement*: It can be used to remove an element from the game; for example, in case of defeating an opponent element. This action is irreversible and, in order to remove the element temporarily, it must be hidden.
– *CreateElement*: This action is used to create a new element on the board. The four latter actions, i.e., *HideElement*, *ShowElement*, *DeleteElement* and *CreateElement* actions cover all the needed operations to be applied on elements in the game.
– *MoveElementToTile*: This action is used to move an element from a tile to another. To perform this task, the unique address of the element and the addresses of the initial and final tiles are needed.
– *DisableTile*: Every tile on the board has an *isEnable* property that shows whether the tile is active or not. In the tile is inactive, it cannot be used to store an element. To make a tile inactive, this action is used.
– *EnableTile*: This is the exact opposite to the *DisableTile* element and is used to activate a tile.
– *NonMovableElement*: All elements have an *isMovable* property that describes whether the element is movable or not. This action is used to deactivate the movement of an element.
– *MovableElement*: This action is the exact opposite to the former action and is used to activate an element's movement.
– *UpdateElementUI*: Every element can be identified on the board using its image. If needed, the image of the element can be changed using this action. This would result in a change in the appearance of the element on the board.

- *UpdatePlayerBoxInfo*: This action can be used to change the text displayed by the player boxes in the top or bottom part of the screen.
- *PlaySound*: This action is used to play a soundtrack or audio in the game, thus resulting in making the game more attractive. In MAndroid, it is possible to play sounds/audios during the game.
- *EndGame*: This action is used for the end of the game. The moment the game reaches the *EndGame* action, the final window is displayed and the name of the winner is announced. After this action, game processing is not possible anymore and the game comes to an end.

***ExtraAttribute*:** The player, element and tile components contain a number of predefined properties that cover all the qualities needed to define the structure of a board game. Besides, there is the possibility for defining new properties during the game. This is achieved by the *ExtraAttribute* component.