Model Driven Development of Board Games

[a]Dogan ALTUNBAY, [b]Eser CETINKAYA, [c]M.Gokhan METIN

Department of Computer Engineering, Bilkent University, TURKEY

[a]altunbay@cs.bilkent.edu.tr, [b]cetinkay@cs.bilkent.edu.tr, [c]m_gokhan_metin@yahoo.com

Abstract Increasing the structural complexity of the video games development process, and the short deadlines which are forced by the market dynamics require to improve the productivity in terms of quality, time, and cost. In this paper we describe a model driven software development approach in order to aid in computer games design and development to address these goals. As an example, we define the related concepts of the board game domain, and provide a board game metamodel both from scratch and using UML profiling mechanism.

Keywords game development, metamodeling, game ontology

1. INTRODUCTION

Game development has grown in complexity and quality, highlighting the need for rapid and mass game software production. For addressing these needs, the developers rely on formal software development processes such as waterfall or agile methodologies. Without leaving these methodologies aside, raising the abstraction level from the "solution domain" to the "problem domain" and using the model-driven based approaches is useful in order to enhance the productivity [1].

In this paper, we propose a metamodel-based development approach for the board game domain. Board games are relatively easier to develop compared with the other types of games. They do not require complicated graphical interfaces, or complex AI rules. In order to demonstrate the model-driven software development processes in the game domain, board game is a suitable choice.

We introduce a meta-model and a number of related concepts, including *GameEngine*, *Player*, *Rules* and etc.. We also provide a Domain-Specific Language (DSL) grammar to express the board game domain effectively. DSLs provide a good basis for domain-specific formal analysis and fully-automated tool support [2].

After presenting the metamodel and related sub-components such as abstract and concrete syntax, we provide two example board game model which are derived from our board game metamodel.

And finally, we come up with the necessary model-to-model and model-to-text transformations in order to have the final working source code for a sample part of a chess game.

The rest of the paper is organized as follows. Section 2 presents a brief domain analysis of the game domain and the board game domain. Section 3 provides the glossary of the domain concepts. Section 4 describes the metamodeling process of the domain. Section 5 gives the concept mapping for the domain. Section 6 explains the UML profiling for the board game domain. Section 7 and 8 describes the model to model and model to text transformations. Section 9 concludes the report. Section 10 gives the references.

2. BOARD GAME DEVELOPMENT

Game generation and development process requires automated systems with intelligent design of games, and reasoning about both the abstract rule system of the game and the visual realization of these rules [3]. There are many researches on developing these generation systems. These researches differ according to their basic strategies. Agent-based meta-modeling systems [4] and development frameworks [5] can be given as examples.

Identifying the content and composing the rules of a game is the starting point of the game design process. Today, computer games are becoming more complex and establishing the correct relationships between different application domains is vital in this process. Inside the video game designing processes, the necessary disciplines which are needed to be integrated might be:

- Game dynamics
- Visualization
- Software programming
- Production phase
- Sound
- Choreographic structuring
- etc.

This divide-and-conquer strategy reduces all the complexity of game design process. The complex nature of video game development arises because of interdependencies between these design elements and process of proper combination of them. Some of the decisions made in one area cause to create different

constraint in another one. For instance, specifications of visual arts can conflict with any technical constraint or the design might appear consistent whereas building it would be totally impractical.

At this point, performing a clear and detailed domain analysis will become beneficial for the game software developers.

Domain analysis is the process of identifying the relevant concepts of an application domain, focusing on the reuse of these concepts. The products of the analysis process are the reusable definitions of the domain concepts that are common for any application of the domain. The methodology of the analysis process may vary depending on the application domain. However, the aim of the process remains the same.

A good domain analysis for the board game domain starts with asking a general question: *"What is the game?"*. After answering this question, we can reduce the domain of *"Game"* to our sub-domain *"Board Game"*.

A game is *'a physical or mental competition conducted according to rules with the participants in direct opposition to each other'* [6]. According to this definition, there are three main components of a game:

- players who are participating the game and in rivalry against their opponents,
- rules which define the constraints of the game, and
- goals that are to be reached by the players by obeying the rules of the game.

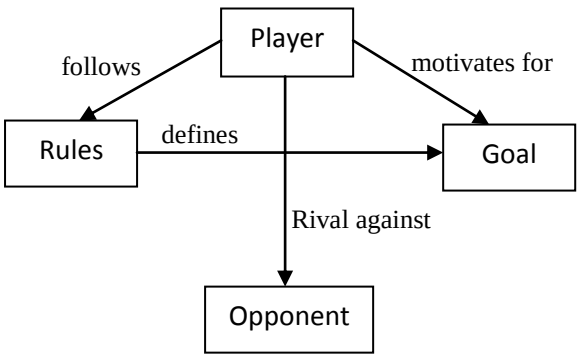

Figure 1: Concepts and relationships of a game.

Figure 1 illustrates the main concepts of the game domain. These concepts are abstract for all types of games. To derive more concrete and specific concepts, we must decrease the abstraction level of the game domain.

In the board game domain, some other concepts that are additional to the abstract concepts of the game domain must be taken into consideration. For example, let's consider the game of Chess. The players must follow the rules which regulate the positions and the movements of each of the chess piece, the state of the player, the timing constraints of the game, and so on.

According to this explanation, additional concepts for the board game domain might be action, game state and game elements.

In our work, we have identified several domain-specific concepts for the board game domain. Namely, these concepts are

- GameEngine,
- GameElement,
- Player,
- Event,
- Action,
- GameState,
- Goal,
- Sub-Goal,
- Non-MovableElement,
- MovableElement, and
- Rules.

Player, Goal, and Rules concepts are inherited from the Game domain. The others are specific for the Board Game domain.

In the following section we describe the these concepts which are obtained as the result of our domain analysis process.

3. GLOSSARY OF DOMAIN CONCEPTS

This section describes the domain concepts of board game applications and relations between these concepts.

GameEngine: A game must have a GameEngine which is responsible for running the game based on the defined rules of the game. Inside the GameEngine, the rules of the specific game should be defined. A GameEngine would have multiple rules according to the domain of the game. It would have a single or multiple states inside which represents the current condition of a specific game. GameEngine has at least one player or more. Moreover, GameEngine should have one or more GameElements and manipulates them during the game play. GameEngine would have All instances of GameEngine must have a board in order players to be able to play game on.

GameElement: The GameElement metaclass represents all the objects inside a specific game. All GameElements are obtained by a GameEngine. They are the artifacts of game. It has two types; a game element would either be a movable or non-movable. Moveable GameElements are the ones which a player can manipulate by creating an action. Non-

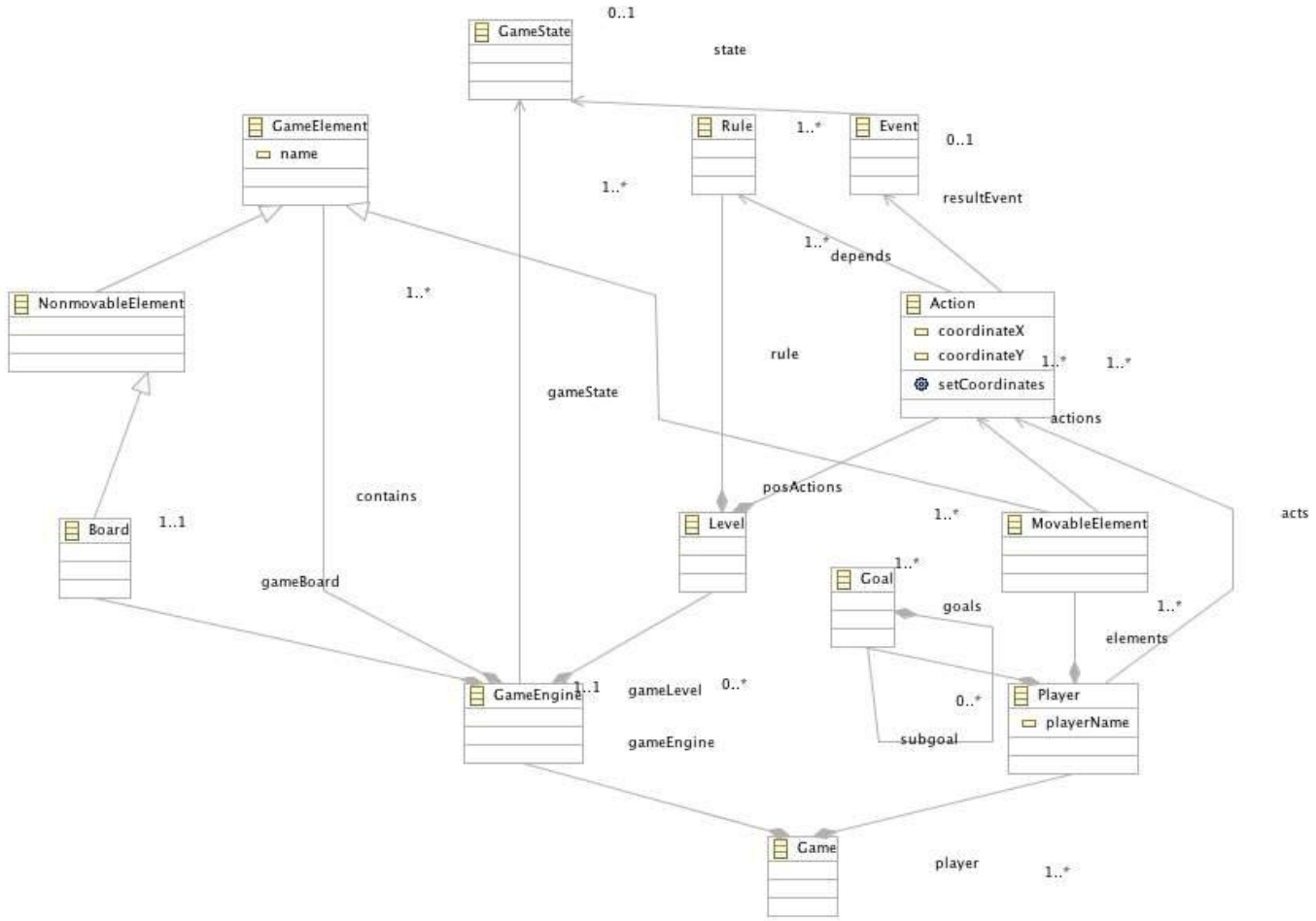

Figure 2 – Board Game Metamodel

moveable GameElements are the ones which cannot be manipulated by any player. In some games, GameElements can change state from moveable to non- movable or vice versa according to the rules of games. For instance, in chess any element which is eliminated by rival player becomes a non-movable element.

Player: Players are the decision makers inside a game. Players have Movable elements which are manipulated via Player Actions. A player can create 0 to n Actions during the game. Some of the Actions could change the state of any Movable Element while some of the Actions do not have effects on. In a game at least one player must exists. Each Player has goals, objectives and an external environment with which they interact.

Event: Event is a condition in which the opponents Actions are restricted. Some of the player actions would case an event to occur for opponent player. The conditions of Events and Actions should be defined by Rules of Game in detail.

Action: Actions are the movements of Players. A Player would have multiple actions during the game. Actions can change the State of game via creating events. Actions would also be able to manipulate the Game Elements. Player makes decisions and applies them to the game by creating Actions.

GameState: GameState metaclass represents the current condition of the game at any instance. In all of the board games, state of the game should be defined. GameState of the game can be changed by Player Action or Event.

Goal: In any board game, goal is the state which players try to attain by creating Actions on Game Elements. In a game, desired goal can be achieved by either completing all the Sub-Goals or just by completing itself. Any global Goal would have some sub-goals which are all part of completing winning process. Actions of player would cause any goal/sub-goal to be completed or not.

Sub-Goal: Sub-Goal concept is defined such that in some games it is required to achieve global goal by completing its parts in order. Goals would have some smaller sub-goals. In such cases, Player has to achieve all the sub-goals in order to reach the global Goal.

Non-MovableElement: Non-MovableElements are the ones which cannot be manipulated by any player by an action. These are the static-artifacts of all games. Generally they are game-specific elements. In board-games the most

common non-moveable object in games is the board on which all the movements of GameElements are performed.

MovableElement: MovableElements are the ones which can be manipulated by any player. Each Player may control one or more Movable Elements via Actions. In some games

certain MovableElement have unique designations and capabilities such as chess. In some games Movable Elements have same capabilities such as Backgammon. Moreover, in some games MovableElements may not belong to a particular player such as Clue.

Rules: Rules are the constraints that define how to set up a system before playing, relationship between the game and the player of the game. In addition, relationship between the Game Elements and Player are defined vie game rules. Players have to obey the game rules. All the Actions and their effects on GameElements are defined by the Rules. Rules are game specific concepts which generally determine turn over, the rights and responsibilities of players. Rules of the games would change according to the current level of game. All the player actions should be based on the game rules.

4. METAMODEL

Metamodel of a domain is formed of abstract syntax of the domain, static semantics and concrete syntax. Abstract syntax describes the relevant concepts of the domain and the relations between these concepts, which are performed in the previous section. Figure 2 represents the abstract syntax of board game application domain by the help of UML notation. Here, each domain concepts is mapped to a metaclass, which is the instantiation of the MOF elements in meta-metamodel level. The relations between these concepts are shown via associations, reflecting the description of the domain in the previous section.

Static Semantics

Static semantics of a metamodel defines the well-formedness rules of it. These well-formedness rules are used for both defining constraints on how models can be formed, and validating the models constructed upon a specific matamodel. For example, we may need to constraint that for each game model based on board game metamodel must have an element named Board. Or we may need to validate that each level of the game should consist of a different set of rules.

Object Constraint Language(OCL) is a standard way of defining rules in both metamodeling(M2) and modeling(M1) levels. In figure 3, we give a possible set of constraints in M1 level.

```
context GameEngine

        inv:not self.Board.oclIsUndefined &
                        self.Board.size() = 1

 context Player

         inv:self.movableElement.size() >= 1

 context GameEngine

        inv: level.size() >= 1

 context Goal

        inv:self.reject(g | self.subgoals.exists(self = g))

context Level

    inv:self.reject(g | self.rules = g.rules)
```

Figure 3 – Static Semantics Samples

Concrete Syntax

The concrete syntax is used to represent all the domain concepts visually, which are identified and described in the abstract syntax.

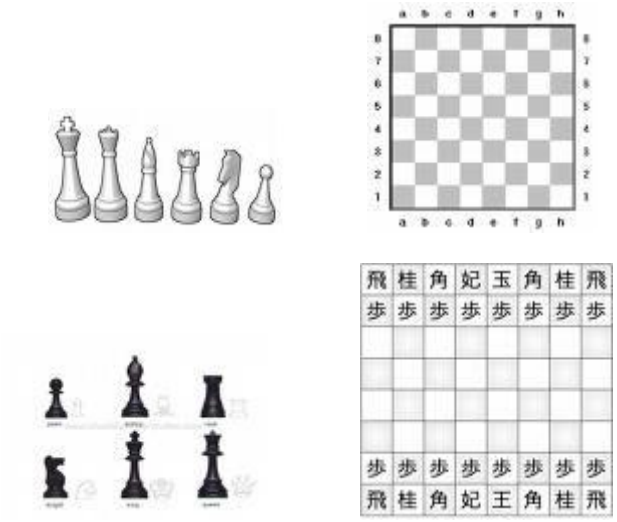

Figure 4: A concrete syntax for the chess pieces and a chess board.

A possible concrete syntax for the chess pieces and the chess board is presented in Figure 4.

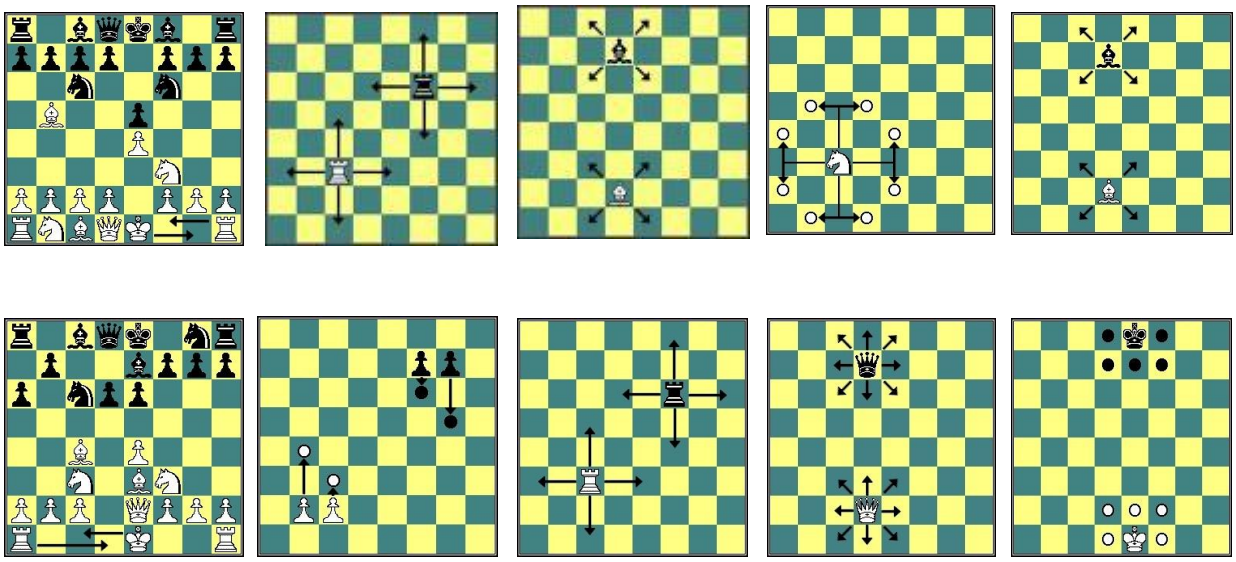

Figure 5: A concrete syntax for the chess game rules.

The same concept can be visual-represented differently in different board games. For example MovableElement, Non-MovableElement and Rule concepts representations for the chess game are shown in Figure 4 and 5.

But these representations cannot be used for another board game, for example the backgammon game. To create a common concrete syntax representation for all board games, UML notation would be appropriate. An example UML concrete syntax is given in Figure 6.

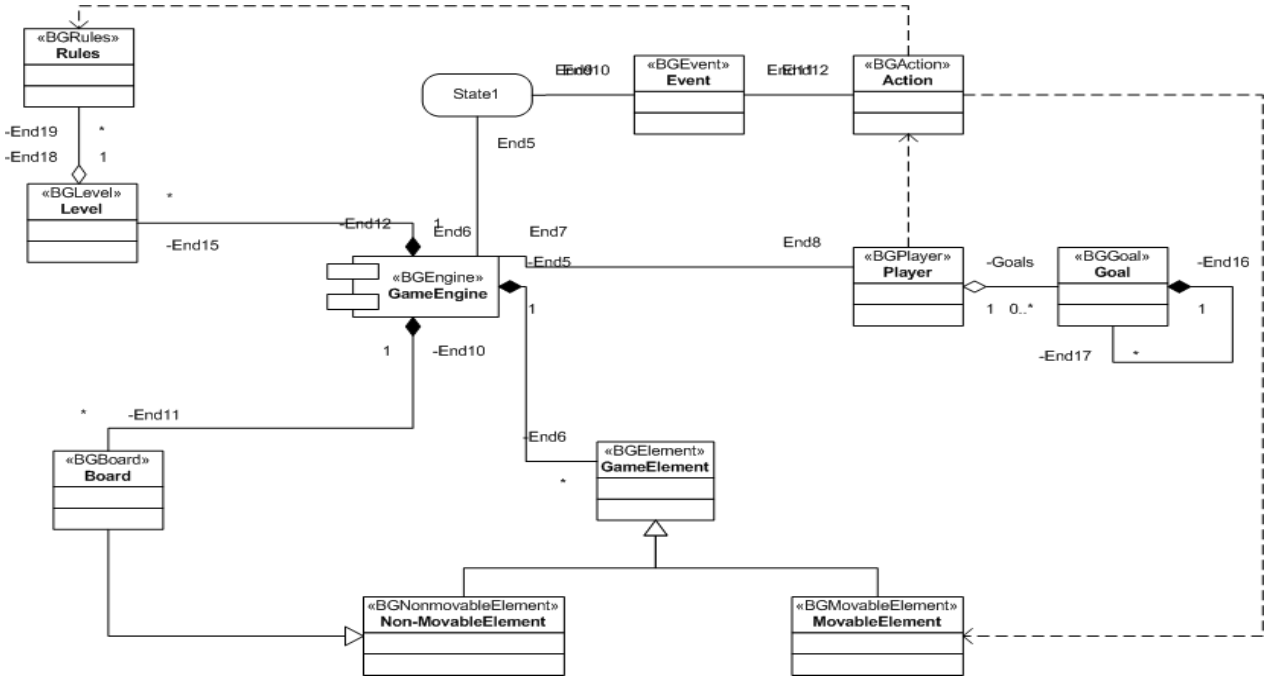

Figure 6: A UML concrete syntax for the chess game.

5. CONCEPT MAPPING

Backus–Naur Form (BNF) is a formal notation used to describe the syntax of a given language. In another word; BNF is a formal way to describe formal languages.

In computer science, BNF is used for specifying the syntax of programming languages, mapping of domain specific concepts to domain-specific grammar in metamodelling process, communication protocols, and similar other things.

On the other hand UML gives the opportunity to extend the UML metamodel in order to define domain specific modeling languages. There exist a number of profiles standardized by OMG for particular domains, including System on Chip, Software Radio, etc.

The extension mechanism of UML allows modeler to define stereotypes and introduce tagged values to them in a formal way. Using profiling mechanism of the UML 2.*, we redefine our metamodel of board game applications. Table 1 lists the stereotypes introduced with this extension process.

```
<Game> :: = <GameEngine> , <Rules> , <Player>;

<Rules> ::= <text>

<GameEngine> ::= <GameElement> , <Rules> , <Level> , <Goal> | <GameElement> , <Rules> , <Goal>;

<GameElement> ::= <MovableElement> | <NonmovableElement>;

<GameElement> ::= <GameElement>,<MovableElement> | <GameElement>,<NonmovableElement>;

<MovableElement> ::= <Token> , <Action> , <VisibleElement> | <Token> , <Action> , <InvisibleElement>;

<VisibleElement> ::= <Token>;

<InvisibleElement> ::= <Timer>;

<NonmovableElement> ::= <Board> , <Player> , <ScoreBoard>;

<Player> ::= <State> , <MovableElement> , <Goal>;
```

Figure 7 – Board Games Grammar

Figure 7 shows the mapping of the domain concepts to a domain specific grammar, which is defined using BNF. We have experienced that BNF has lack of expressing power of all domain-specific concepts. It does not provide the same flexibility as the metamodelling does. We believe that this inability is raised from that BNF is designed to express only the context-free grammar, but not domain-specific context.

To give an example for this situation that in our concept mapping process, BNF was insufficient to express the relationship between the concepts of action and event.

6. UML PROFILE FOR BOARD GAMES

Metamodeling process is a struggling process considering the definition of abstract syntax of the metamodel, concrete syntax which realizes the abstract syntax, and the static semantics of the metamodel. In the previous sections we described the problem domain and defined a metamodel using MOF metalanguage, which is a language for defining metamodels.

Figure 8 shows the UML profile that we define for board game application domain.

Game Model Element	Stereotype	UML Metaclass
GameEngine	BGEngine	Component
GameElement	BGElement	Class
Player	BGPlayer	Class
Event	BGEvent	Class
Action	BGAction	Class
GameState	BGState	State
Goal	BGGoal	Class
Board	BGBoard	Class
MovableElement	BGMovableElement	Class
NonmovableElement	BGNonmovableElement	Class
Rules	BGRules	Class

Table 1 – UML Profile for Board Games

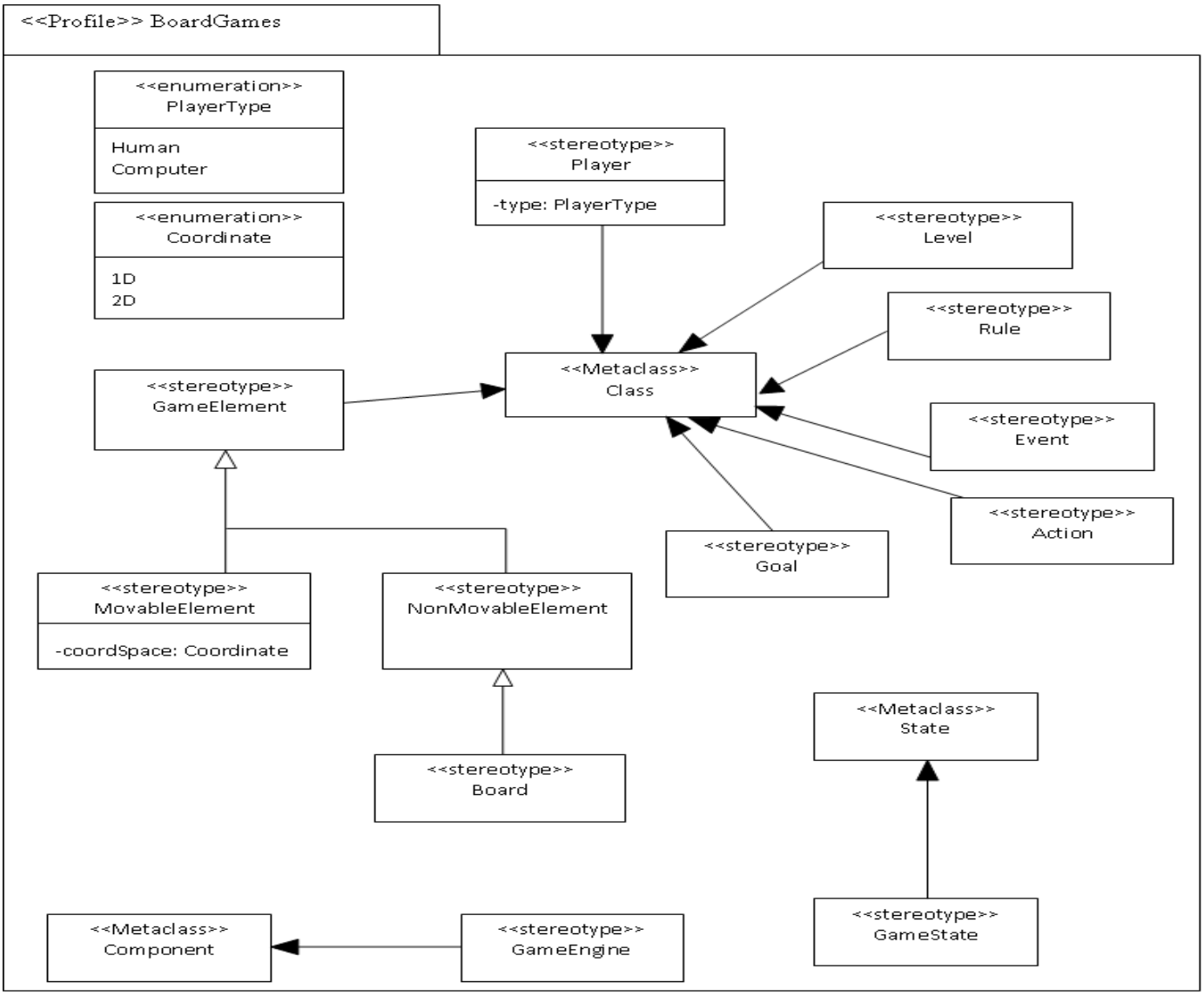

Figure 8 – UML Profile for Board Games

Sample Models

Having defined UML Profiling mechanism for Board Games we provide two example instances for it. First one is a M1 level model for Chess Game. In the Chess domain the instances for metaclass "Rule" corresponds to the Chess rules such as Castling, En Passant and Promotion. These are special rules for chess domain. For the metaclass Movable element there exist six instances such as Pawn, Bishop, Knight, Rook, King, Queen which are shown in Figure 9. For metaclass "Non-MovableElement" ChessBoard and Timer classes are instances. In the chess domain a Player`s action can threthen opponents kings, for this scenerio our example model have a "Chech" class which is an instance of "Event" metacass.

In the second example we created a model for the game Backgammon which is shown in figure 10. In the Backgammon domain we show only two rules for practical reason which are "CollectChecker" and "BrokenChecker". These classes are instances of metaclass "Rule". In this case there exist only one kind of movable element different from Chess which is Checker. In Backgammon domain class "Dice" and "Board" is an instance of "Non-MovableElement" metaclass. There exists two instances for metaclass "State" which are "CheckerState" and "PlayerState". In this context, a players action would create and event "BrokeChecker" as shown in figure 10.

7. MODEL TO MODEL TRANSFORMATIONS

One of the key issues of Model Driven Software Development is interoperability. There exist a number of modeling tools in the market which are based on different metamodels. Ability to port a model definition from one designer tool to another one comes out as an essential requirement.

Our metamodel of Board Games, as described in the previous sections, is a sub domain of Games domain. Thus it contains only the concepts and relations between them which are required for only board games. On the other hand, Games domain is considerably large and complex compared to our domain model. Figure 14 shows GameDSL[9] metamodel, which is a domain model of video games. As can be seen in the figure, the GameDSL metamodel contains a number of additional concepts with respect to our Board Game metamodel. Thus, the GameDSL metamodel is more complex compared to our metamodel.

By means of interoperability, the need of mapping sub domain concepts to super domain concepts comes out as a problem. To this end, we aim to transform our model instances to models conforming GameDSL metamodel in order to achieve general game models.

Model to model transformations, which is a basic concept of MDSD enables us to perform this mapping of different metamodels. Once transformations between metamodels are defined at metamodel layer, model instances can be transformed to instances of the target metamodel. Then we can benefit from the tools which implement the target metamodel. Furthermore, we can enhance our models using these tools.

Atlas Transformation Language – ATL

ATL[7] is a model transformation language which is developed at INRIA and provided within the Eclipse Modeling Project[8]. ATL transformations are based on transformation rules which define the mappings between target and source metamodel elements.

Table 2 gives the mappings between our Board Game metamodel and GameDSL metamodel. Transformation rules are listed in Appendix.

BoardGame Element	GameDSL Element
GameEngine	Game
MovableElement	ActiveEntity
NonMovableElement	StaticEntity
Board	ContainerObject
Level	Level
Action	Action
State	State
Event	Event

Table 2 – Mappings between BoardGame vs GameDSL domain models.

8. MODEL TO TEXT TRANSFORMATIONS

The Model to Text transformation plays a key role in the Model-Driven based software development process. It addresses how to translate a model to various text artefacts such as code, deployment specifications, reports, documents, etc.

Essentially, the model to text standard needs to address how to transform a model into a linearized text representation. An intuitive way to address this requirement is a template based approach wherein the text to be generated from models is specified as a set of text templates that are parameterized with model elements. The OMG "MOF Model to Text Transformation Language RFP" aims to achieve a standard technique for this task.

The model to text transformation process is based on parsing the model structure outputting the desired code. As an example to this process is working with the MOFscript tool.

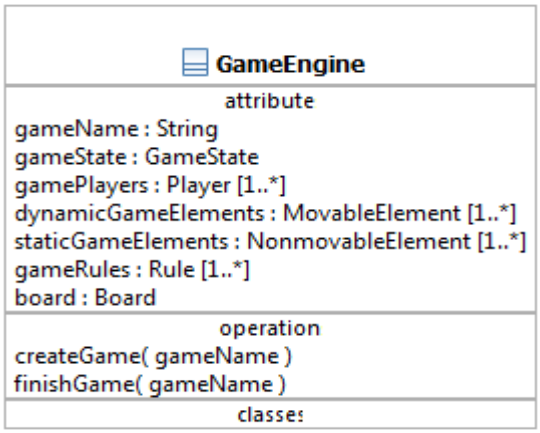

Figure 11 – GameEngine class.

```
/**
* Default method constructor
*/
uml.Class::methodConstructor() {

  self.ownedOperation->forEach(o:uml.Operation) {
    o.visibility ' '
    if (o.returnResult.size() > 0) {
      '' + o.returnResult.first().type.name ''
    }
    else {
      'void '
    }
    o.name '('
    if (o.ownedParameter.size() > 1) {
      o.ownedParameter->forEach(g:uml.Parameter) {
        if
(g.name.equals(o.ownedParameter.last().name)) {
          g.name
        }
        else {
          g.name', '
        }
      }
    }
    else {
      o.ownedParameter->forEach(g:uml.Parameter) {
        g.name
      }
    }
      ') {
  }
```

Figure 12 – MOFScript template for code generation.

```
public class GameEngine {

	/**
	 * Attributes
	 */

	String gameName;
	GameState gameState;
	Player gamePlayers;
	MovableElement dynamicGameElements;
	NonmovableElement staticGameElements;
	Rule gameRules;
	Board board;

	/**
	 * Class constructor
	 */

	public GameEngine () {
	}

	/**
	 * Methods
	 */

	public void createGame(gameName) {
	}

	public void finishGame(gameName) {
	}


} //End of class GameEngine
```

Figure 13 – Generated GameEngine.java file.

To work with this tool, the models have been exported to the Eclipse Modeling Framework (EMF) format, and using the MOFscript specific language, the desired text outputs have been produced. We have used this tool to generate a java file for each of the class in the chess game model. Figure 11 illustrates the GameEngine class structure. In Figure 12, a part of the transformation template (methodConstructor) is shown. The generated GameEngine.java file is given in Figure 13.

In this project, we also used model to text transformation technique for creating a textual representation for the chess game by using OpenArchitectureWare (oAW) tool. oAW is a modular MDA/MDD generator framework implemented in Java. It supports parsing of arbitrary models, and a language family to check and transform models as well as generate code based on them.

The oAW framework is based on the EMF, which again, is based on the eCore meta-modelling language. ECore Meta-Models constitute the abstract syntax for our model.

We created our chess game model in XML form which conforms the game metamodel that is defined in an XML schema (XSD). After completing this phase, a transformation template is developed by using XPAND language. The related code and result of this transformation process is placed in Appendix.

9. CONCLUSIONS

Model Driven Software Development is a comprehensive process that enables a high level software development methodology by encapsulating the low level processes from developers with simplified and domain oriented definitions. As a part of this process, metamodeling has considerably big amount of importance, and should be essentially focused on.

On the other hand, with respect to immature status of modeling tools the modeling process becomes considerably struggling. Especially the transformation phase of the process required large amount of effort. In this study we used Eclipse Modeling Framework, which is the most widely used platform for modeling. However, large number of Eclipse plug-ins provided for modeling are incubation releases that have unresolved bugs, and these bugs makes it difficult to focus on the modeling process.

In this paper we described a domain analysis for board games and proposed metamodels for the domain based on MOF and using extension mechanism of the UML metamodel. During this process we find out that defining concrete syntax from scratch is a difficult task and UML concrete syntax may be used instead. The following step in the model driven development of board games is defining model-to-model and model-to-code transformations.

10. REFERENCES

[1] Reyno, E.M., Cubel, J.A.C., *Model-Driven Game Development: 2D Platform Game Prototyping*.

[2] Brucker, A.D., Doser, J., *Metamodel-based UML Notations for Domain-specific Languages*, 4th International Workshop on Language Engineering (ATEM 2007), pp. 1-??, 2007.

[3] Mark J. Nelson, Michael Mateas, *Towards Automated Game Design, In AI*IA 2007: Artificial Intelligence and Human-Oriented Computing*

[4] Steve Goschnick, Sandrine Balbo, Liz Sonenberg, *ShaMAN: An Agent Meta-model for Computer Games.*

[5] Robin Hunicke, Marc LeBlanc, Robert Zubek, *MDA: A Formal Approach to Game Design and Game Research, In Proceedings of the Challenges in Games AI Workshop, Nineteenth National Conference of Artificial Intelligence.*

[6] Merriam-Webster. Game. Merriam-Webster Online Dictionary, accessed Apr. 16, 2009. http://www.merriam-webster.com/dictionary/game.

[7] ATL Official Site: http://www.eclipse.org/gmt/atl

[8] Eclipse GMT Official Site: http://www.eclipse.org/gmt

[9] GameDSL Official Site: http://gamedsl.tuxfamily.org/

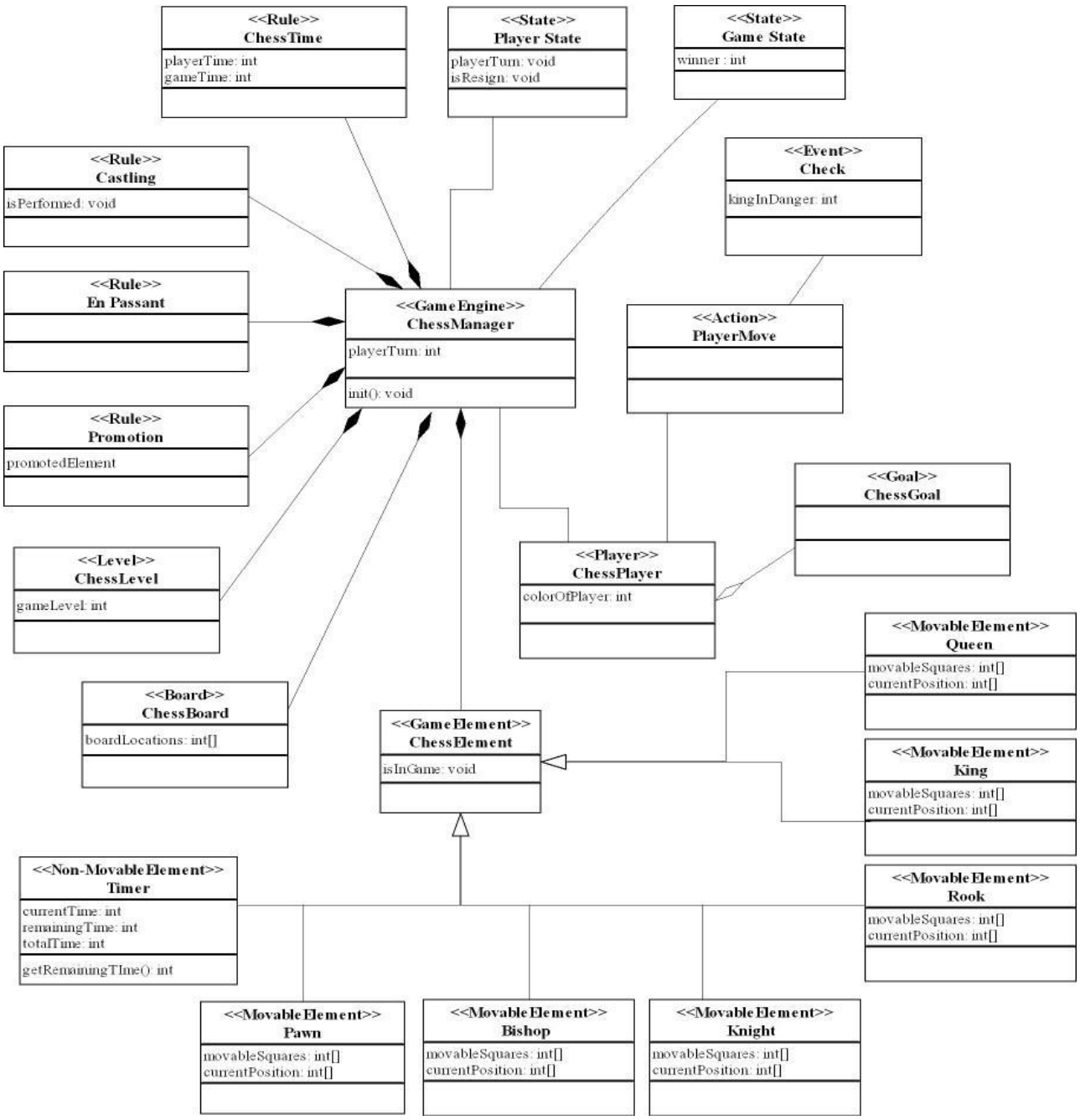

Figure 9 – Chess Game sample model

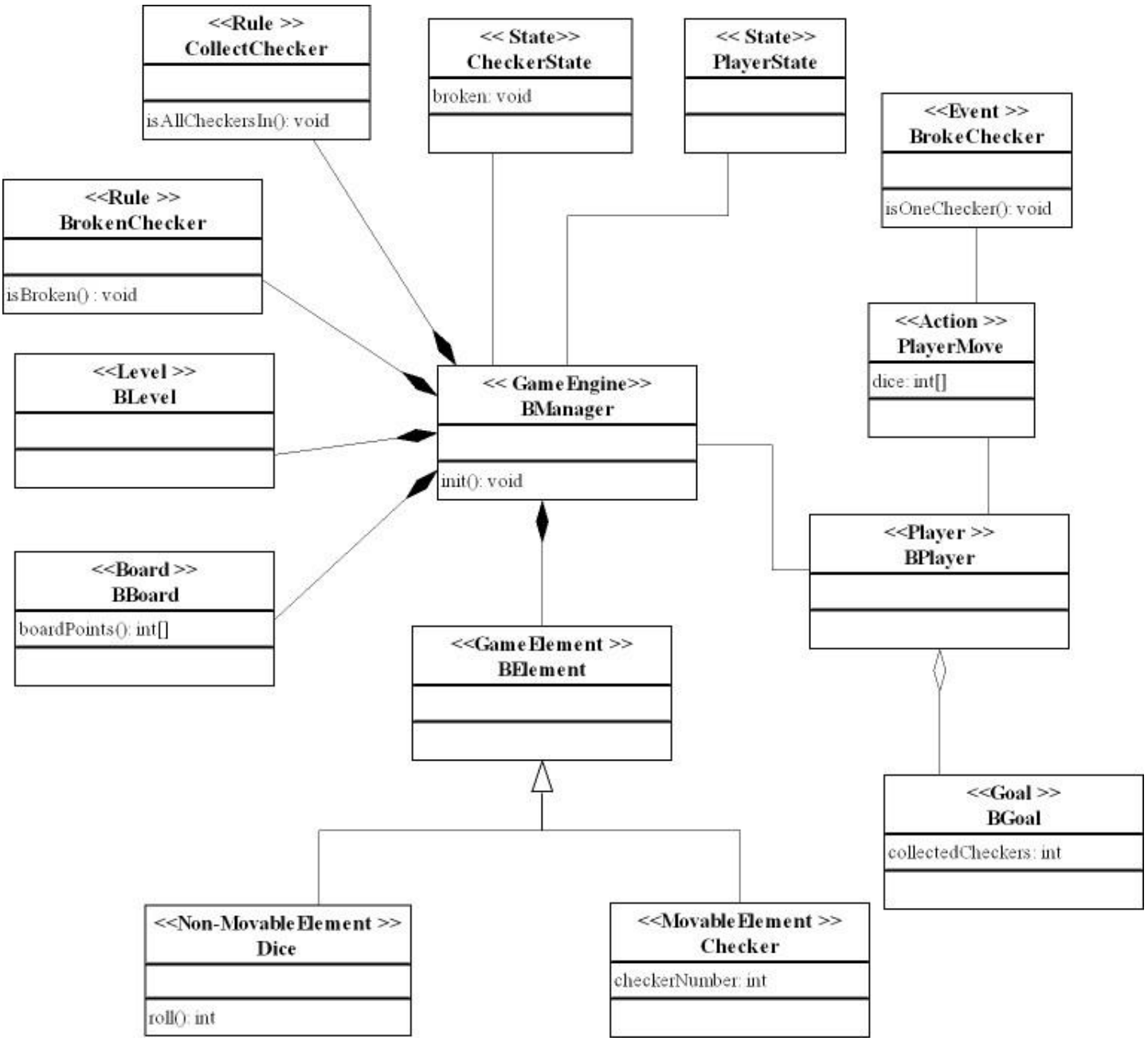

Figure 10 – Backgammon sample model

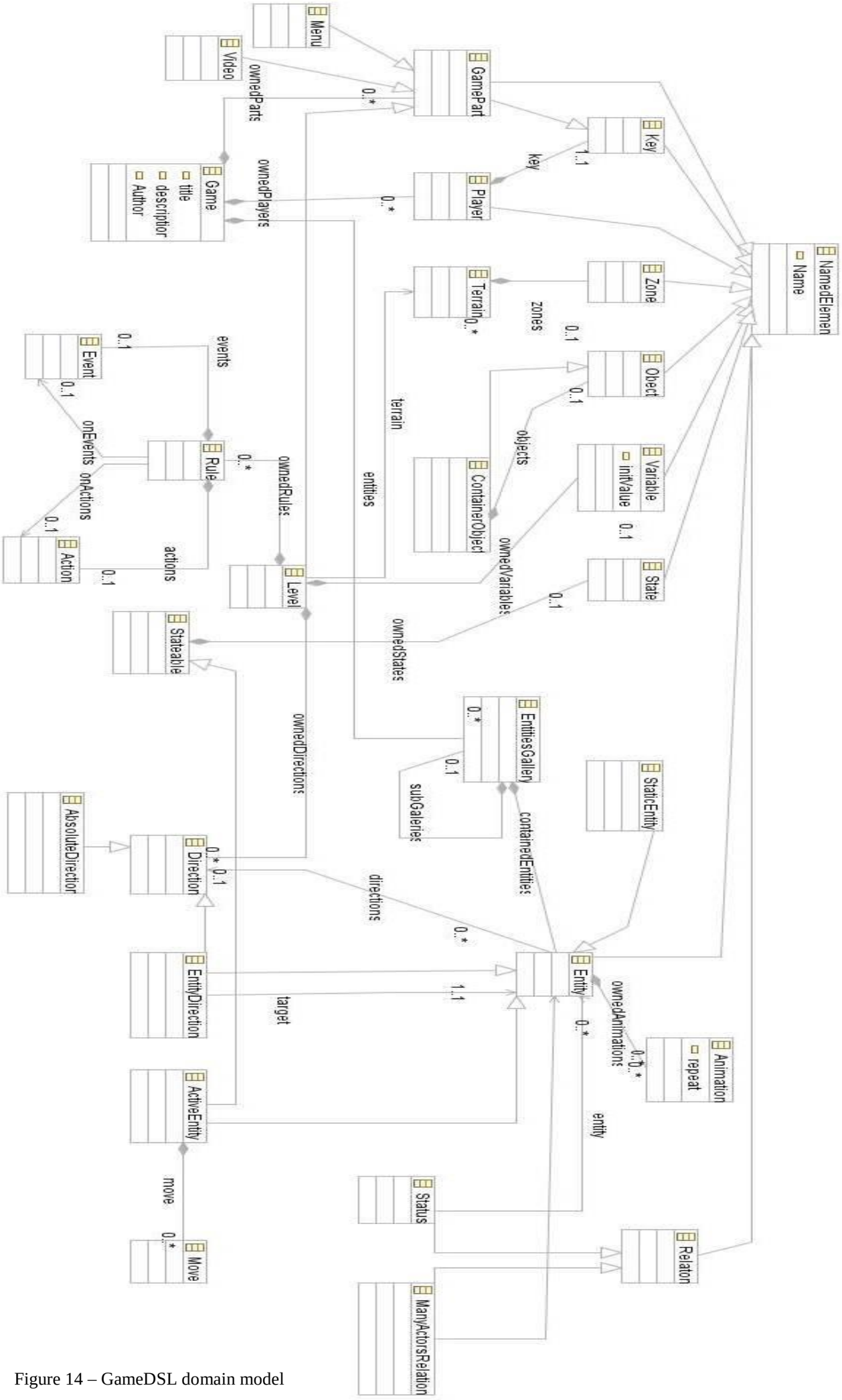

Figure 14 – GameDSL domain model

Appendix A – Code Listings

Model to Text Transformations

GAME METAMODEL DEFINED IN XML SCHEMA (METAMODEL.XSD)

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:tns="http://www.example.org/game" elementFormDefault="qualified"
targetNamespace="http://www.example.org/game">

  <complexType name="Game">
    <sequence>
       <element name="start" type="IDREF"/>
       <element name="gameEngine" type="tns:GameEngine"/>
    </sequence>
  </complexType>

  <complexType name="GameEngine">
    <sequence>
       <element name="gameName" type="string"/>
       <element name="gameElement" type="tns:GameElement"/>
       <element name="nonmovableElement" type="tns:NonmovableElement"/>
       <element name="movableElement" type="tns:MovableElement"/>
       <element name="board" type="tns:Board"/>
       <element name="player" type="tns:Player"/>
     </sequence>
  </complexType>

  <complexType name="GameElement">
    <sequence>
       <element name="gameElementType" type="string"/>
    </sequence>
  </complexType>

  <complexType name="NonmovableElement">
    <sequence>
       <element name="name" type="string"/>
    </sequence>
  </complexType>

  <complexType name="MovableElement">
     <sequence>
       <element name="name" type="string"/>
    </sequence>
  </complexType>

  <complexType name="Board">
    <sequence>
       <element name="xCoordinates" type="string"/>
       <element name="yCoordinates" type="string"/>
    </sequence>
  </complexType>

  <complexType name="Player">
    <sequence>
       <element name="playerName" type="string"/>
       <element name="action" type="tns:Action"/>
       <element name="goal" type="tns:Goal"/>
  </sequence>
  </complexType>
```

```
  <complexType name="Action">
    <sequence>
       <element name="coordinateX" type="string"/>
       <element name="coordinateY" type="string"/>
    </sequence>
  </complexType>

  <complexType name="Goal">
    <sequence>
       <element name="subGoal" type="tns:Goal"/>
    </sequence>
  </complexType>

</schema>
```

CHESS GAME MODEL DEFINED IN XML FILE (MODEL.XML)

```
<?xml version="1.0" encoding="UTF-8"?>
<game xmlns="http://www.example.org/game">
  <start>Chess Game Metamodel Structure</start>
  <gameEngine>
    <gameName>Chess Game</gameName>
    <gameElement>
       <gameElementType>Movable Elements, Nonmovable Elements</gameElementType>
    </gameElement>
    <nonmovableElement>
       <name>Board, timer</name>
    </nonmovableElement>
    <movableElement>
       <name>king, queen, rooks, bishops, knights, pawns</name>
    </movableElement>
    <board>
       <xCoordinates>a, b, c, d, e, f, g, h</xCoordinates>
       <yCoordinates>1, 2, 3, 4, 5, 6, 7, 8</yCoordinates>
    </board>
    <player>
       <playerName>Player Name</playerName>
       <action>
          <coordinateX>Player's move x-coordinate</coordinateX>
          <coordinateY>Player's move y-coordinate</coordinateY>
       </action>
       <goal>
          <subGoal></subGoal>
       </goal>
    </player>
  </gameEngine>
</game>
```

MODEL-TO-TEXT TRANSFORMATION TEMPLATE IN XPAND LANGUAGE (TEMPLATE.XPT)

```
«IMPORT metamodel»

«DEFINE Root FOR metamodel::Game»
«FILE "ChessGame.str"»
Explanation: «start»
Game Name: «gameEngine.gameName»
```

```
Game Element Types: «gameEngine.gameElement.gameElementType»
Non-movable Element Types: «gameEngine.nonmovableElement.name»
Movable Element Types: «gameEngine.movableElement.name»
Board X-Coordinates: «gameEngine.board.xCoordinates»
Board Y-Coordinates: «gameEngine.board.yCoordinates»
Player Name: «gameEngine.player.playerName»
Player's Action X-Coordinate: «gameEngine.player.action.coordinateX»
Player's Action Y-Coordinate: «gameEngine.player.action.coordinateY»
«ENDFILE»
«ENDDEFINE»
```

Model to Model Transformations

```
module BoardGame2GameDSL; -- Module Template
create  OUT: GameDSL from  IN: BoardGame;

rule GameEngine2Game{
	from
		ge: BoardGame!GameEngine
	to
		g: GameDSL!Game(
			title <- 'Game', Author <- 'DEG', description <- 'Generated sample
chess game'
		)
}


rule MovableElement2ActiveEntity{
	from
		me: BoardGame!MovableElement
	to
		ae: GameDSL!ActiveEntity(
			name <- me.name
		)
}

rule NonMovableElement2StaticEntity{
	from
		nme: BoardGame!NonMovableElement
	to
		ne: GameDSL!StaticEntity(
			name <- nme.name
		)
}
```