This article has been accepted for inclusion in the International Journal of Gaming and Computer-Mediated Simulations, 2(2), 18-35, April-June 2010.
Content is final as presented, with the exception of layout and pagination.

Decoupling Aspects in Board Games Modeling

Fulvio Frapolli, Amos Brocco , Apostolos Malatras and Béat Hirsbrunner
Department of Informatics, University of Fribourg, Switzerland

Existing research on computer enhanced board games is mainly focused on user interaction issues and look-and-feel, but overlooks the flexibility of traditional board games when it comes to game rule handling. In this respect, we argue that successful game designs need to exploit the advantages of the digital world as well as retaining such flexibility. To achieve this goal, both the rules of the game and the graphical representation should be simple to define at the design stage, and easy to change before or even during a game session. For that reason we propose a framework allowing the implementation of all aspects of a board game in a fully flexible and decoupled way. In this paper, we describe the FlexibleRules approach, which combines both a model driven and an aspect oriented design of computer enhanced board games. The benefits of this approach are discussed and illustrated in the case of three different board games.

Introduction

Recent research in the domain of multi-user interaction, such as multi-touch interactive tables (Loenen et al., 2007; Mazalek, Reynolds, & Davenport, 2007), opened a broad range of new possibilities redefining the concept of human-computer interaction. Noteworthy applications of these devices are computer enhanced games, which take advantage of both the physical and the digital worlds in order to improve the user experience. Our research is focused on computer enhanced board games, aiming at improving user experience by mixing the full flexibility of traditional board games played around a table with computational functionalities from the digital world.

Games benefit from the features offered by digital environments, such as the high degree of dynamicity that can be introduced by means of advanced visual and audio effects. These further promote improvements in the immersive experience (Amory & Adams, 1999), and the interactivity of the game-play (Malone, 1981). It is evident that a significant amount of benefit can be gained by transferring the concepts and the games themselves from the physical to the digital world. Advanced visualization capabilities spur the development of innovative and sophisticated representations of game graphics and computer support can also help ease complex game tasks or situations, for instance by calculating intricate winning conditions, or by performing mundane tasks such as card shuffling or point distribution. However, the rules that guide the game-play are typically handled by the game software and are tightly intertwined with it, with their implementation hidden and inaccessible during game-play.

While the porting of physical board games to their computer-enhanced counterparts has been to a large extent successful, there exist certain aspects of traditional game-play that are not inherently supported to date. These deficits diminish the merits of computer-enhanced board games and lead to players registering a smaller degree of game satisfaction. As pointed out in (DeKoven, 1978; Salen & Zimmerman, 2003), the ability modify the rules should not be considered just as an additional feature of the game, but as a central aspect of it that should not be neglected. It empowers the players by giving them overall control of the game and its features, while at the same time enabling them to modify the level of difficulty of the game or even its winning conditions. One additional advantage of being able to dynamically update the game rules and logic is the ability to extend the game-play and incorporate or update specific options and parameters that are usually hardcoded in the game software. Unfortunately, traditional approaches to game software development fail to support this vision, and represent a high barrier for both casual and experienced players without any programming skills, wanting to modify some rule of the game. Furthermore, depending on how the game is implemented, it could be difficult even for a programmer to add a certain rule without having to modify large portions of the code. In contrast, physical games allow the redefinition of rules by means of social agreement between players at any time during game-play.

In this paper we propose an extensible and efficient framework called FlexibleRules that aims at taking advantage of both approaches (i.e. physical and digital), by allowing the implementation of board games in a fully flexible and decoupled way. The FlexibleRules framework is comprised of both a conceptual model to design board games and also a set of tools, including a domain-specific language and a dedicated compiler, to realize the aforementioned design. The different aspects of the game, such as the logical behavior of the different game objects, their representation, and the outcome of each action are modeled separately and can be freely modified during game-play. The main goal is to promote modularity and clarity: the user should be able to quickly identify what is to be modified and where in order to change something in the game. Another requirement that was taken into consideration was simplicity of use, as it is not to be expected that all users will be skilled developers. To this end, the FlexibleRules framework employs a user-friendly, Lisp-inspired language to implement its functionality. Another important aspect is providing the user with full control of

This article has been accepted for inclusion in the International Journal of Gaming and Computer-Mediated Simulations, 2(2), 18-35, April-June 2010.
Content is final as presented, with the exception of layout and pagination.

the degree of automation: game rules can either be enforced by the system, or left to a human referee. Additionally, the framework aims at providing game designers with tools supporting the modeling of a game and allowing the creation of prototypes that can be tested and fine tuned.

The rest of this paper is organized as follows. The next section discusses and reviews related work in the field of aspect oriented development and games. We then present the conceptual model that lies behind the FlexibleRules framework, followed by a detailed description of the basics of the game definition language. The development environment that enables the implementation of actual board games based on the aforementioned model and language is subsequently described. In the section concerning the FlexibleRules examples, we review the implementation of three different games, illustrating their differences in the light of the FlexibleRules framework. Finally, we provide some conclusions on the presented work and insights on future work, having first discussed relevant implications in research, development and practice.

Related Work

The motivation behind this research work has been to fully support the conveying of physical board games into the digital environment. The main deficit of existing digital board games is that they take into account the graphical representation of the game and its rules of play, but neglect the social interactions that occur during traditional sessions of game-play. One aspect of this is the freedom of letting players define house rules to make the game more enjoyable and suited to their standards. It has been well-established (DeKoven, 1978; Salen & Zimmerman, 2003) that this ability constitutes a focal point of any successful game design. In order for the game to have any degree of flexibility, interactivity and be enjoyable to play, the support for house rules (Mandryk & Maranan, 2002) is a necessity, since amongst other things, it promotes a much desired level of human-to-human interaction. In this respect, we present a holistic framework, called FlexibleRules, to address the aforementioned issues at the game modeling level.

The need to establish standard models for the design and subsequent analysis of games is evident, as it allows for a common understanding and a shared vision among developers and also users (Bjork & Holopainen, 2004). Modeling a board game requires a deep understanding of all the objects involved, their behaviors and interactions, as well as the laws that govern the game world (i.e. rules that define allowed actions during game-play) and the winning conditions (Sanchez-Crespo, 2003). A first effort to simplify the modeling of a problem can be achieved through object orientation, and the definition of objects roles (Steimann, 2000). By recognizing the different entities composing the game, along with their relationships and roles, it is possible to create a model in a more natural way. Additionally, by separating different functional concerns encompassing single entities, as done with Aspect Oriented Programming (AOP) (Kiczales et al., 1997), the logic behind each object can be further simplified. Whereas (Steimann, 2005) considers aspects only as concerns of programming, there are many examples presenting them as core functional parts of a system (Rashid & Moreira, 2006), and as useful abstraction mechanisms which help express computer programs in a more natural way (Lopes, Dourish, Lorenz, & Lieberherr, 2003). In the same spirit, we want to promote aspects as a central functional part of modeling, and argue that an advanced separation of concerns will not only ease the modeling and implementation phases (Miller, 2001), but also effectively help end-users better understand the logic of the game.

A first step in this direction was taken by (Reese, Duvigneau, Köhler, Moldt, & Rölke, 2003), who divided the logic behind the game in terms of states, behaviors and rules. Different aspects of the game Settlers Of Catan™ are distributed between two types of agents: administrative agents and players. Administrative agents are in charge of controlling the game board state, monitoring the enforcement of the game rules, and the resources owned by player agents, while player agents represent both the computer controlled player and the human counterpart. (Järvinen, 2003) proposes an approach for describing game rules as different aspects. The authors point out that game rules can be divided into five types: rules governing the game components and their function, rules governing relations between elements, rules that define game environments (i.e. the physical boundaries of components), rules that define the theme and rules for the user interactions.

Building on the aforementioned concepts and ideas we plan to extend current approaches and distribute all of the game logic within the game entities, as well as dividing it into different aspects; by clearly separating concerns of the game, we aim at simplifying the logic, and easing its comprehension by users, while catering for the full support of flexibility that traditional board games exhibit.

FlexibleRules Model

FlexibleRules is a framework for modeling and implementing board games around their atomic elements, the game entities, aiming at a simplified functional description of the game logic and its graphical representation. This section details all the elements that conceptually define the game model, both from a structural and a functional point of view. The game definition language presented in the following section comprises the infrastructure for developers to implement their games based on the proposed conceptual model.

We recognize two levels of abstraction as far as game modeling is concerned, namely logic and representation, which provide us with an initial separation of concerns. These concerns are modeled separately as logical and representation layers, dealing with a low-level description of game dynamics, and the high-level interface with the real-world (typically a graphical or tangible representation) respectively. Thus, the game model can be viewed as the composition of these two layers.

This article has been accepted for inclusion in the International Journal of Gaming and Computer-Mediated Simulations, 2(2), 18-35, April-June 2010.
Content is final as presented, with the exception of layout and pagination.

Logic Layer

The Logic Layer is comprised of entities, which are the building blocks that define the functional core of a game. We distinguish between two types of logic entities: active (such as human or computer players) and reactive (such as the pawn or the board). Reactive entities behave in response to events originating from active ones. Because the FlexibleRules framework is not concerned with modeling real or artificial intelligence players, the proposed model only focuses on reactive entities, and for the rest of this paper we will use the term entity as referring only to reactive ones. Extending the proposed model to encompass active entities will be considered in our future research. An overview of the elements of the FlexibleRules model concerning the logic layer and the interactions between these elements is shown in Figure 1.

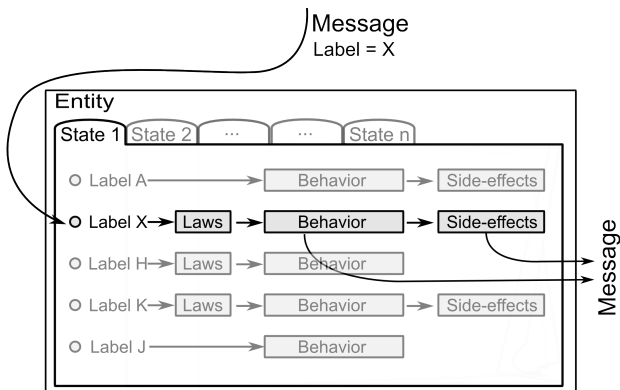

Figure 1. Overview of the FlexibleRules model

An entity is characterized by a functional behavior and a set of private properties, which can be accessed and modified only by the entity itself. Coordination and communication between entities is performed through the use of messages. Messages are information containers that are exchanged between entities during execution. Each message is identified by a label, which is used to dispatch it. All game actions are triggered by information exchanged between entities through message passing. Upon reception of a message, an entity triggers a certain internal reaction, its *behavior*, according not only to the received information, but also to its current internal state. The latter is stored as a specific *state* property in each entity. This allows the implementation of separate behaviors for the same message type for different states. The execution mechanism is thus similar to a finite automaton, such that actions executed by an entity are univocally defined by the input triple *entity, state, message label*.

Semantic Aspects. The behavior of an entity describes how it should act in a certain situation, without being concerned with game rules that might disallow that particular behavior or make it produce some secondary outcome. These additional aspects of the game, referred to respectively as *laws* and *side-effects*, are modeled separately as *around* and *after returning* advices.

Laws The join point (a join point is the aspect-oriented programming term for an interaction point with the rest of the system (Elrad, Filman, & Bader, 2001)) for laws is before the execution of a particular behavior: according to the current state of the entity, and the values in the incoming message, preconditions for the execution of the behavior are checked. If pre-conditions are not met, a law advice can prevent the execution of the behavior. Laws also typically prevent the game from reaching an invalid situation, and can perform a rollback to restore the last valid situation (for example, the status just before an invalid player move). In order to analyze the actual global situation, laws can access all properties defined by all the entities of the game.

Side-Effects The join point for side-effects is after the execution of a behavior. Side-effects define secondary outcomes of a behavior (for example, assigning points to the players after the successful completion of a complete round in a turn-based board game). Within a side-effect, it is possible to query attributes of each game entity, perform rollbacks, and send messages to other entities (for example, to inform them about the points earned by the player in the current turn). Finally, side-effects can also be used to determine whether winning conditions have been reached, thus ending the game session.

Representation Layer

Having defined the logic layer and the semantic aspects of the entities that comprise it, we present in this section the second level of abstraction that we have previously introduced, namely that of representation. Logic entities may have a dual in the representation layer, commonly referred to as their *representation*. Different forms of representation can exist, such as graphical or tangible. On a computer interface, a representation is typically composed of some graphical elements that decorate the game world and provide a visual feedback of the actual situation in the game. At any time during the game-play, a representation should reflect the internal state of its logical counterpart, by accessing its properties and executing any appropriate update procedures to mirror any potential changes in the background logic. Moreover, representations can also define representational properties, typically for storing graphical aspects such as their color, size or their position on the screen. To retain consistency with the proposed aspect-oriented approach throughout the FlexibleRules framework, updates to the representations are also modeled as separate aspects in the form of *before* or *after* advices.

Representation Updates Join points for representation updates are placed before or after changes to internal properties of a logical entity. In this respect, the *behavior* of the representation layer is to observe the logical layer and respond to any modifications by appropriately reformatting the graphical representation of the game. Update procedures can access both the value before and after the property change, therefore allowing the implementation of advanced graphical transitions.

This article has been accepted for inclusion in the International Journal of Gaming and Computer-Mediated Simulations, 2(2), 18-35, April-June 2010.
Content is final as presented, with the exception of layout and pagination.

User Interaction

A game is a system that often requires mixed user interaction. On one hand the user should be able to interact explicitly with the game by manipulating representations (*Intentional User Interaction*), e.g. to move pawns around a checkerboard; on the other hand an entity may request user intervention in order to perform a certain operation (*Forced User Interaction*), e.g. information cards presented to the users to notify them about game state. These two different aspects of user interaction concern both the logical and the representation layers: logic entities may trigger forced interaction, whereas representations may be the starting point for intentional interaction.

Intentional User Interaction: the representation becomes an interface that allows the user to communicate with game entities. During the game process, a representation can send messages to its logical counterpart, and can also provide a visual feedback by modifying its appearance.

Forced User Interaction: interactions with the user can be necessary to determine the sequence of events to be executed in the game. Thus, the execution of a behavior may temporarily stop to allow interaction with the user, and then continue accordingly. Forced user interaction can be compared to modal message dialogs shown by computer applications.

Game Definition Language

The core of the FlexibleRules framework is the domain-specific programming language for representing and subsequently developing board games, based on the conceptual model that we have presented. The language exposes all the previously presented abstractions and allows the full implementation of different kinds of board games using the FlexibleRules development environment that will be presented in the following section.

Language Basics

The language syntax is inspired by Lisp: expressions are written using a prefix notation, and are enclosed within parentheses. It is noteworthy to mention that we have also implemented a compiler for the language we defined. The need to introduce a novel language is spurred from the observation that current scripting languages have a general scope, while we wished to express the semantics of the FlexibleRules model and hence defined this domain-specific languages specifically targeted at board games design.

The language uses dynamic typing, and recognizes six different data types: numbers (either integer or float values), strings, lists, dictionaries (hashtables mapping string keys to data values), messages and *nil* (equivalent to Boolean false). Common operators to manipulate these data types are available. Local temporary variables, with scope limited to the current behavior, can be defined using the **var** statement:

(var ⟨*identifier*⟩ ⟨*value*⟩**)**

Flow control can be managed both by using conditional statements, as well as by means of loop statements:

(if ⟨*condition*⟩ ⟨*body*⟩ **else** ⟨*body*⟩**)**

(switch ⟨*identifier*⟩
  (case ⟨*value*⟩ ⟨*body*⟩**)**
  . . .
  (case ⟨*value*⟩ ⟨*body*⟩**))**

(foreach ⟨*identifier*⟩ **in** ⟨*list*⟩ ⟨*body*⟩**)**

(while ⟨*condition*⟩ ⟨*body*⟩**)**

The *body* element shown in the previous examples represents only a single statement. To define sequences of multiple statements the following expression can be used:

(do ⟨*statement*⟩ **. . .** ⟨*statement*⟩ **end)**

Entity Definition

Entities are composed of behaviors and properties. For each possible state, we can define the behavior to be executed upon receiving a message with the specified label as follows:

#state ⟨*name*⟩
  #onMessage ⟨*label*⟩
    ⟨*behavior*⟩
  . . .
  #onMessage ⟨*label*⟩
    ⟨*behavior*⟩

An entity can modify its state at any time by means of the **changeState** function. Properties can be defined and initialized from within a behavior using the following statement:

(new property ⟨*name*⟩ **as** ⟨*value*⟩**)**

Values associated with properties can be retrieved using the **property** function, and changed with the **update property** function as follows:

(property ⟨*name*⟩**)**

(update property ⟨*name*⟩ **to** ⟨*value*⟩**)**

Notice that the **property** function does not return a reference to the property but just its value. To manipulate property values it is thus often necessary to use a temporary variable and update the property afterwards.

Messaging

Entities can communicate by exchanging messages, which carry key-value pairs called *attributes*. Messages are identified by a label, which is itself an attribute associated with the key *label*. The primitive for sending messages along with their attributes is the following:

This article has been accepted for inclusion in the International Journal of Gaming and Computer-Mediated Simulations, 2(2), 18-35, April-June 2010.
Content is final as presented, with the exception of layout and pagination.

(send new message ⟨*label*⟩ **to** ⟨*recipient*⟩ **with**
⟨*attribute name*⟩ ⟨*value*⟩
. . .
⟨*attribute name*⟩ ⟨*value*⟩**)**

It is also possible to change some values of the received message and then forward it:

(update message attribute ⟨*key*⟩ **to** ⟨*value*⟩**)**

(forward message to ⟨*recipient*⟩**)**

The sending or forwarding of messages is a blocking operation, i.e. the current behavior is halted until the recipient completes its execution. An entity waiting on a send or forward operation can nonetheless accept and process incoming messages.

Laws and Side Effects

Law and Side-Effect advices are defined using the same syntax as behaviors, i.e. the aforementioned language statements. To bind them with an entity, the following directives are used:

#law on ⟨*label*⟩ **to** ⟨*entity*⟩ **in** ⟨*state*⟩ **is** ⟨*law*⟩

#sideeffect on ⟨*label*⟩ **to** ⟨*entity*⟩ **in** ⟨*state*⟩ **is** ⟨*se*⟩

Assuming that laws and side-effects are stored in different files, these directives bind the law specified in file *law*, respectively the side-effect in file *se*, to entity *entity*. In particular, the law is applied upon receiving a message with label *label* in state *state*, whereas the side-effect executes after the corresponding behavior. If multiple laws and side-effects match a specified join-point, they are executed sequentially following the order of the definitions. It is possible to refer to generic pointcuts by replacing either *label* or *state* with the wildcard character $*$. For example,

#law on * to ⟨*entity*⟩ **in "default" is** ⟨*law*⟩

inserts a law advice before every behavior in state *default*. Laws and side-effects can also cancel a player's move by performing a rollback. In order to perform a rollback, an entity must first create a checkpoint to save the actual status of game:

(open checkpoint)

A rollback can then be invoked to restore the status up to the last checkpoint:

(rollback ⟨*reason*⟩**)**

The *reason* argument allows to specify a text message to be shown to the user explainaining the reason for the rollback.

Representations

Representations are the entities' interface to the real world. Since actual implementation just focuses on 2*D* representations, we restrict the world to a 2-dimensional plane displayed on a computer screen, but other kind of interfaces could be easily implemented. Thus, the representation of an entity consists of one or more graphical objects, each one characterized by some *physical* attributes, such as size, position on the canvas, color, etc. According to the FlexibleRules model, a representation must reflect the status of the corresponding logical entity at all times. Each representation therefore, as we have already described, defines update procedures that are executed upon modification of any of the observed properties of the underlying logical entity. Additionally, representations can also define private properties not tied to their *physical* appearance, in order to keep persistent information across updates.

A representation is modeled similarly to an entity; nonetheless some important differences exist:

- representations can instantiate and manipulate graphical objects, whereas entities cannot;
- representations can only send messages to the underlying entity (for user interaction purposes).

The latter difference results in a tight relationship between the representation and the corresponding logical entity.

Graphical Objects. Representations can create simple graphical objects on the representation canvas, which can be identified by means of a unique name. These objects can only be modified or destroyed by the representation. To create a new graphical object, the **new graphic** function is used in the following manner:

(new graphic ⟨*id*⟩**)**

(new graphic ⟨*id*⟩ **with**
⟨*attribute name*⟩ ⟨*value*⟩
. . .
⟨*attribute name*⟩ ⟨*value*⟩**)**

Attributes of the graphical object can be specified at construction time (using **with** ⟨*attributes*⟩), as well as modified at runtime as follows:

(update graphic ⟨*id*⟩ **attribute** ⟨*attribute*⟩ **to** ⟨*value*⟩ **)**

Finally, it is possible to delete a graphical object using the **delete graphic** function.

Representation Updates. One noteworthy element of the language is the coupling of the representation layer to the logical one. Representations can define observers on properties defined by their logical counterpart. Observers are triggered when the corresponding property is updated, and can be effectively set up by the representation as follows:

(observe ⟨*property*⟩ **notify** ⟨*label*⟩**)**

Before the observed property is updated to a new value, the representation receives a notification message labeled *label* along with the new value.

This article has been accepted for inclusion in the International Journal of Gaming and Computer-Mediated Simulations, 2(2), 18-35, April-June 2010.
Content is final as presented, with the exception of layout and pagination.

User Interaction

User interaction with graphical objects can be distinguished to signals and messages. On one side, Intentional User Interaction is accomplished by hooking interaction events generated by graphical objects to the delivery of predefined messages that are dispatched to the representation. As an example, to deliver the message *message* when the user clicks on the object *id*, the syntax is:

(on ⟨*id*⟩ **clicked notify** ⟨*message*⟩**)**

Notice that the message must be constructed in advance, and can carry additional user-defined attributes. On the other side, Forced User Interaction is achieved with *signals*. Signals are generated by graphical objects in response to some user action; an entity waits on a signal using the **wait signal** primitive, stopping its execution until the signal is emitted by some representation.

(on ⟨*id*⟩ **clicked emit** ⟨*signal*⟩ **with** ⟨*tag*⟩**)**

(wait signal ⟨*signal*⟩**)**

Upon hooking an event with a signal, a string *tag* value may be specified: when the corresponding wait function intercepts the signal, this value will be set as the return value.

Development Environment

The FlexibleRules framework is founded on the aforementioned well-defined conceptual model and additionally provides a development environment taking into account the separation of concerns and enabling rapid implementation of computer enhanced board games. The implementation of board games using the development environment is a two step process. The first step involves the *Logic Editor* tool that allows high-level design of the game building on the principles of the model, namely defining the corresponding entities, their properties and relations in a visual manner. During the second step the actual implementation of the game functionalities takes place through the *Code Editor*.

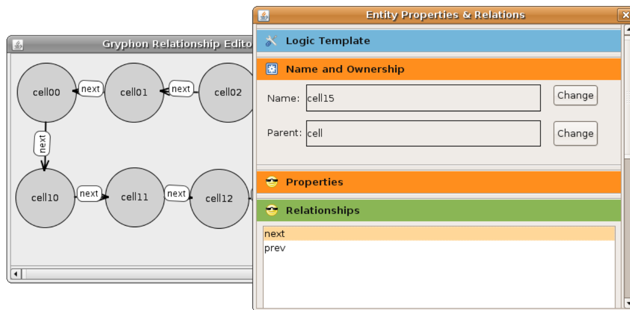

Figure 2. Logic Editor

The *Logic Editor* (Figure 2) is used to visually define game entities along with their properties and relationships, allowing therefore for a user-friendly modeling of the logical structure of the game. In the editor, relationships among entities are represented by means of edges of a directed graph, the nodes of which are the game entities. Moreover, the designer is given the option to define a hierarchy of entities, thus enabling inheritance of entities' characteristics and their local properties. This creates the abstract outline of the logical part of the entities. Concrete implementation of the game itself is achieved by implementing entities' functionalities, such as the specification of *behaviors*, *laws* and *side-effects*. The latter is performed using the *Code Editor*, which is based on the open source Gedit[1] text editor that has been enhanced with custom plug-ins, such as syntax highlighting of the Game Definition Language. As depicted in Figure 3, the left side of the *Code Editor* enables the browsing of code, while in parallel maintaining an overview of its structure. Furthermore, in the lower part a pane displaying the order of *laws* and *side-effects* attached to a specific *behavior* is presented.

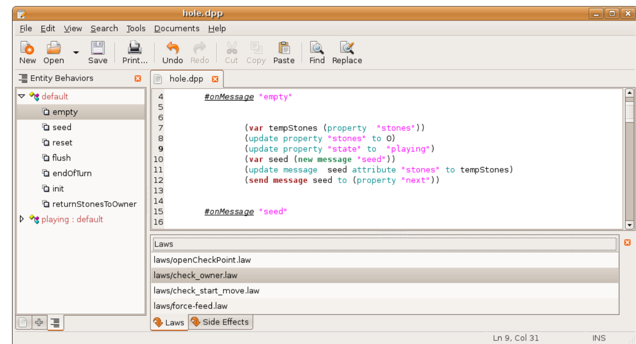

Figure 3. Code Editor

FlexibleRules Game Examples

We have presented the conceptual model of the FlexibleRules framework and explained the Game Definition Language that we have implemented along with the two visual tools composing the framework, namely the *Logic Editor* and the *Code Editor*. The Java programming language was used to implement the tools in order to take advantage of its portability and interoperability across diverse platforms. The latter constituted an important requirement for our design, as we envisage that the developed board games will be deployed on a multitude of hardware infrastructures, such as PCs, interactive tables, PDAs, etc. In the following we present some of the games that have been implemented to validate the FlexibleRules framework approach. In particular, we strive to highlight the major conceptual differences between these games, and thus the different aspects involved in their modeling. Table 1 provides a summary of these characteristics.

The games that we chose to illustrate as proof-of-concepts, available to play online at the FlexibleRules website[2], are Awele, Go and Himalaya™. Awele is one of the

[1] http://projects.gnome.org/gedit/
[2] http://diuf.unifr.ch/pai/flexiblerules

This article has been accepted for inclusion in the International Journal of Gaming and Computer-Mediated Simulations, 2(2), 18-35, April-June 2010.
Content is final as presented, with the exception of layout and pagination.

Table 1
Game Models Characteristics Comparison

	Awele	**Go**	**Himalaya™**
Types of Entities	few	few	many
Board Topology	Ring	Grid	Free
User Interfaces	single (public)	single (public)	multiple (public, private)
Number of Players	2	2	3 to 6
Mundane Tasks	None	None	Many
Game-play and Rules Complexity	Simple	Medium	Hard
User Interaction	Start of turn	Start of turn	Continous
Game Variants	Many	Few	Few

games with the biggest number of known variants in the world, which validates the need to have a framework like the proposed one in order to allow for the modification of rules and thus cater for multiple variants. The difficulties in designing Go in a fully distributed manner, make it a great candidate to exhibit the capabilities of our framework in this respect. The game logic and ruleset of Himalaya™ have a higher degree of complexity compared to the other two games. Taken together, the implementation of these three games serves as evidence that the FlexibleRules framework can be equally successfully used for different types of board games.

Awele

Awele (also known as Oware) is an African turn-based game (Figure 4). It is an interesting example because in contrast to other popular games, there exist hundreds of different variants. We recognize very few entities in the game model, namely *hole* (which manages its seeds and the actual game-play), *game* (which manages player turns and players' points earned during the game). The basic behavior of each entity is easily modeled, and most of the different variants of the game can be implemented by just adding or removing laws and side-effects. Since many *laws* and *side-effects* require a deep knowledge of the underlying structure of the game, in the following we will discuss only a simple example of both a law and a side effect regarding allowed moves and winning conditions. Table 2 summarizes all the *laws* and *side-effects* attached to the entities *hole* and *game*.

The most recurrent rule in turn based games such as Awele dictates that a player cannot play unless it is her turn (Figure 5). The implementation just compares the current player property stored by the *game* entity with the identifier of the player executing the move.

A side effect that can be easily changed is the number of points required to win the game. Commonly a player wins by collecting at least 25 stones. The validity of this condition is checked every time a hole sends its stones to the game by the *Check Winning Condition* side-effect (Figure 6). A potential variant of the game would just require a change in the value being tested.

It is interesting to note that this winning condition is completely separated from the behavior of the entities. In fact, it could be possible to completely remove this side effect, thus letting the users continue their game.

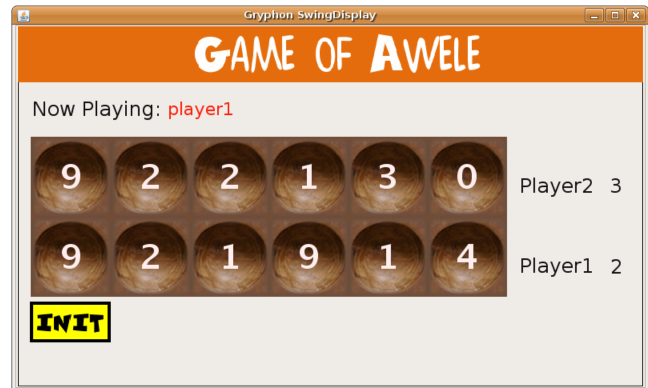

Figure 4. Awele Game

```
(var currentPlayer (first (property "players" of "game")))

(if (!= (property "owner") currentPlayer)
        (return))
```

Figure 5. Law: Allow Only Current Player

```
(var currentPlayer (first (property "players")))
(var CurrentPlayerStones (match currentPlayer in (property "playerStones")))

(if (>  CurrentPlayerStones 24) (do
        (send new message "gameover" to "game" with
                "winner" currentPlayer)
end))
```

Figure 6. Side-effect: Check Winning Condition

Go

Go (Figure 7) falls in the category of checkerboard games and is probably one of the most interesting game that we have chosen to model and implement with the FlexibleRules framework. First different game variants exist. Secondly, the complexity of interactions between game entities is certainly higher than in Awele. The modeling of the game is challenging and game-play rules can be implemented by the cells themselves. Since each cell is only aware of its local neighbors, many steps of the game-play require the coordination of cells' behaviors. In the example of Figure 8 we illustrate

This article has been accepted for inclusion in the International Journal of Gaming and Computer-Mediated Simulations, 2(2), 18-35, April-June 2010. Content is final as presented, with the exception of layout and pagination.

Table 2
Awele Laws & Side-effects

Law	**Entity**	**Behavior**	**State**
Allow Only Current Player	Hole	startSeeding	default
Allow Only Non Empty Hole	Hole	startSeeding	default
Let The Opponent Play	Hole	startSeeding	default
Open Check-point	Hole	startSeeding	default
Stones' Capturing	Hole	giveStones	default/lastSeed

Side-effect	**Entity**	**Behavior**	**State**
End Of Turn	Hole	startSeeding	default
Last Seed	Hole	seed	default
Chain Reaction	Hole	giveStones	default
Chain Reaction Avoiding GrandSlam	Hole	giveStones	lastSeed
Game Over	Game	endOfTurn	default
Check Winning Condition	Game	giveStonesToCurrentPlayer	default

the logical modeling of a cell and the execution flow occurring when a new stone is placed on it by means of a message labeled "placeStone".

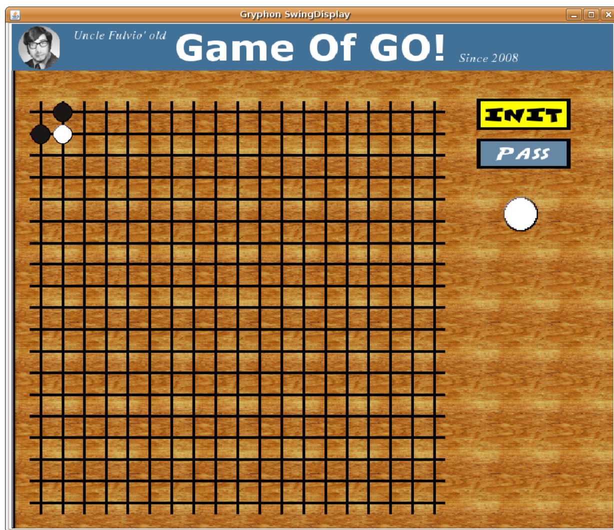

Figure 7. Go Game

At any time in the game a cell can be in one of four different internal states. Empty cells are in state *default*, while cells hosting a stone are in state *withStone*. During the game, cells have to recognize in a distributed way whether they have liberties or not; hence two additional intermediate states are needed, namely *checkingLiberties* to underline that a cell cannot infer its liberties without the help of its neighbors and *captured* for cells that have established that they do not have any liberties. The behavior of a cell when a new stone is placed on it consists in changing its state to *withStone* and storing the stone's color. The rules that are attached to this specific behavior include a *law* and three *side-effects*, as shown in Figure 8. Certain moves are forbidden in Go (e.g. a stone cannot commit "suicide") thus a check-point should be opened allowing for a rollback, if such a forbidden situation is reached. Moreover, the *updateLiberties* side-effect sends messages to the neighbors to trigger the update of their liberties, as the newly placed stone may have removed the last liberty of some opponent's stones and therefore led to their removal from the game. Subsequently, the *checkSuicide* side-effect checks whether the newly placed stone is situated in a cell without liberties. If this is the case the suicide rule has been broken and a rollback to the last valid checkpoint (stored by the law) has to be performed. Otherwise, the final side-effect *endOfTurn* is executed informing all interested entities that the turn is over.

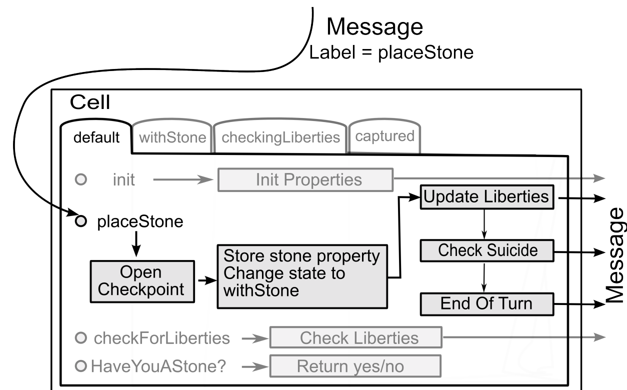

Figure 8. Go modeled using FlexibleRules

Himalaya™

Himalaya™ is a board game characterized by the complexity of its rules. In contrast to Awele and Go, the game can be played by 3 to 6 players, and interactions between the user and the game are continuous. In fact, each player secretly chooses her moves (typically 6) at the beginning of each turn. Once all the players have made their choices, the game proceeds by executing one move per player. During

This article has been accepted for inclusion in the International Journal of Gaming and Computer-Mediated Simulations, 2(2), 18-35, April-June 2010.
Content is final as presented, with the exception of layout and pagination.

game-play some actions require user intervention, for example to let the user choose where to place some objects on the board; thus, Himalaya™ also served as a validation testbed for Forced User Interaction, as conceived within the FlexibleRules framework.

Figure 9. Himalaya™ Game

Figure 10. Himalaya™ Player View

The implementation makes use of different separate displays: a public one, and personal ones for each player. The public display (pictured in Figure 9) serves as the gameboard, which provides visual feedback of the game situation and of players actions. Each personal display (Figure 10) shows private information and enables players to secretly interact with the game while choosing their moves and actions.

FlexibleRules Implications

The FlexibleRules framework makes it easier for game designers and casual gamers to develop new board games or modify existing ones. Regarding game designers, the aforementioned advanced features of FlexibleRules enable rapid prototyping and beta testing of digital board games, having therefore a noteworthy impact on the time and effort required to find the best possible game balance. Contrariwise, casual gamers gain an increased level of satisfaction and enjoyability, as they have overall control of the game and its features.

Furthermore, FlexibleRules has implications on the educational domain as it can be used to stimulate the programming learning process through its user friendly intuitive environment. IT students profit from and are motivated by a game-oriented development environment, which effectively makes the interactive learning of basic programming paradigms more enjoyable and hence successful. Moreover, it is evident that the direct involvement of the end-user in the game development process will lead to enhanced and more widely accepted digital board games.

Conclusions and Future Work

The facilitation of board game development efforts and support for the complete set of features that traditional physical board games exhibit, most importantly the high degree of flexibility in player actions, has been the motivation of our work. In this respect, we deem it necessary to have a well-established model for board game modeling, taking into account their inherent characteristics and avoiding the rigidness of current implementations. Additionally, developers of games should be provided with useful tools to make their task easier and to allow for game-specific patterns to be employed.

In this paper we presented the FlexibleRules framework for the modeling of board games. The proposed game model is composed of a taxonomy of entities with precise properties and behaviors. In order to simplify the conceptualization of rules, we proposed a decoupling of the game logic into different aspects: laws, behaviors, and side-effects. Consistent with the goals of aspect oriented programming, we believe that such a separation of concerns allows for a more natural way to define the logic behind board games. The framework has been validated through the implementation of three games, i.e. Awele, Go and Himalaya™ .

Future work will include the improvement of existing tools and the implementation of an integrated visual game development environment. Additionaly both a qualitative and a quantitative user evaluation of the framework will be carried out. The former aims at assessing the game-play enjoyment compared to traditional physical games, while the latter will focus on the usability of the game modeling language and tools, which will give us indications about the effort put in by inexperienced users to do the modeling.

Acknowledgments

The authors would like to thank ...... for her invaluable help and useful comments.

References

Amory, A. N. K. V. J., & Adams, C. (1999). The use of computer games as an educational tool: identification of appropriate game types and game elements. *British Journal of Educational Technology*, *30*(4), 311–321.

Bjork, S., & Holopainen, J. (2004). *Patterns in game design (game development series)*. Charles River Media.

DeKoven, B. D. (1978). *The well-played game: A player's philosophy*. Anchor Books, New York.

This article has been accepted for inclusion in the International Journal of Gaming and Computer-Mediated Simulations, 2(2), 18-35, April-June 2010. Content is final as presented, with the exception of layout and pagination.

Elrad, T., Filman, R. E., & Bader, A. (2001). Aspect-oriented programming: Introduction. *Commun. ACM*, *44*(10), 29–32.

Järvinen, A. (2003, November). Making and breaking games: a typology of rules. In C. Marinka & R. Joost (Eds.), *Level up conference proceedings* (pp. 68–79). Utrecht: University of Utrecht.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., et al. (1997). Aspect-oriented programming. In *Ecoop'97 object-oriented programming* (pp. 220–242).

Loenen, E. van, Bergman, T., Buil, V., Gelder, K. van, Groten, M., Hollemans, G., et al. (2007). Entertaible: A solution for social gaming experiences. In *Tangible play workshop, iui conference.*

Lopes, C. V., Dourish, P., Lorenz, D. H., & Lieberherr, K. (2003). Beyond aop: toward naturalistic programming. In *Oopsla '03: Companion of the 18th annual acm sigplan conference on object-oriented programming, systems, languages, and applications* (pp. 198–207). New York, NY, USA: ACM.

Malone, T. W. (1981). Toward a theory of intrinsically motivating instruction. *Cognitive Science*, *5*(4), 333 - 369.

Mandryk, R. L., & Maranan, D. S. (2002). False prophets: exploring hybrid board/video games. In *Chi '02: Chi '02 extended abstracts on human factors in computing systems* (pp. 640–641). New York, NY, USA: ACM.

Mazalek, A., Reynolds, M., & Davenport, G. (2007, October). The tviews table in the home. *Horizontal Interactive Human-Computer Systems, 2007. TABLETOP '07. Second Annual IEEE International Workshop on*, 52–59.

Miller, S. (2001, Apr). Aspect-oriented programming takes aim at software complexity. *IEEE Computer*, *34*(4), 18-21.

Rashid, A., & Moreira, A. (2006). Domain models are not aspect free. *Model Driven Engineering Languages and Systems*, 155–169.

Reese, C., Duvigneau, M., Köhler, M., Moldt, D., & Rölke, H. (2003, February). Agent based settler game. In *Proceedings of agentcities agent technology competition (atc03), barcelona, spain.* Agentcities.NET.

Salen, K., & Zimmerman, E. (2003). *Rules of play : Game design fundamentals*. The MIT Press.

Sanchez-Crespo, D. (2003). *Core techniques and algorithms in game programming*. New Riders Games.

Steimann, F. (2000). On the representation of roles in object-oriented and conceptual modelling. *Data Knowledge Engineering*, *35*(1), 83-106.

Steimann, F. (2005). Domain models are aspect free. In L. C. Briand & C. Williams (Eds.), *Models* (Vol. 3713, p. 171-185). Springer.