CoDePA Studio: Adding Explicit Support for Behavior Variants in Authoring Games

Bram Pellens
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
+32 2 629 11 03
Bram.Pellens@vub.ac.be

Frederic Kleinermann
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
+32 2 629 11 03
Frederic.Kleinermann@vub.ac.be

Olga De Troyer
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
+32 2 629 11 03
Olga.DeTroyer@vub.ac.be

ABSTRACT

The development of computer games is both complex and technically challenging, especially when it comes to designing complex behavior for computer games. Current development tools do not provide any high-level design facilities for behavior and require the designer to manually program the behavior. Therefore, the <name> approach was introduced to facilitate the authoring of behavior in computer games (and other interactive 3D applications). This approach uses conceptual modeling techniques to elevate the specification of behavior to a higher level. Code generation from the conceptual specifications is supported. Furthermore, Generative Design Patterns are used to allow reusing existing solutions. In this paper, we explain how the approach has been extended with techniques from the domain of Software Variability, i.e. feature models and configuration models, to support the specification and generation of different flavors of a behavior. In this way, we effectively support a common way of working in game development where one often uses similar behavior scripts except for some variations. By providing support for this at a conceptual level, we make this practice explicit and elevate it to a higher level such that it can be better controlled and exploited. The paper also introduces the design tool developed to support the approach.

Keywords

Conceptual Modeling, Behavior Authoring, Software Variability, Software Product Line, Feature Models, Graphical Specifications

1. INTRODUCTION

Computer games have grown from a niche market into a “multi-billion dollar” industry with a very big economic impact. In 2009, the turnover worldwide for the game industry was more than $50 billion and according to PriceWaterhouseCoopers, it will grow up to $73.5 billion in 2013 [13]. This same report also mentions that cost-effectively developing computer games remains one of the biggest challenges in this field.

The majority of the effort in game development revolves around content creation and many resources are spent on it. Game companies use a set of content creation tools to aid the designers in their work. However, these tools only focus on the artwork, interfaces, game levels, and so on, but none of them really provide support for developing the game story, i.e. the scripts for the behavioral aspects of the computer game. For complex behaviors, the developer still has to resort to manually writing code using scripting languages such as Lua [12] and UnrealScript [4]. Having better ways to support the behavior development for games may lead to a higher productivity and reduce the development cost.

One of our research objectives is to investigate and develop techniques and approaches to decrease the development cost of games and more in particular the development of the behavioral part of games. This research is performed in collaboration with a <country> game development company specialized in Role Playing Games (RPGs). After studying behavior scripts developed by this company for one of their games, we came to the conclusion that they actually reuse as much as possible the same behaviors. For example, characters in an RPG game (and even over games) can be of different species, can play different roles and manage various skills, but a lot of their behaviors are similar. This is reflected in the behavior scripts. We discovered that a lot of scripts were actually very much alike, except from some variations. Often these variations had to do with the characteristics of the characters. This is a form of software variability [3]. However, in this case, the variability is hard-coded into the scripts. This has several disadvantages: it is difficult to maintain the common parts; there is no explicit overview of all possible variants and of reusable parts; and dependencies are hidden in the code. While the purpose of this practice is code reuse, the last two disadvantages mentioned will actual hinder this reuse of code. New or other developers may not be aware of what can be reused without digging into the code. Therefore, we realized that the company could benefit from having this variability knowledge made explicit and by providing them tools to support variability explicitly.

We have developed an approach and supporting tool based on Software Product Lines (SPL) [5] to deal with variability in the development of behaviors. SPL refers to engineering techniques for creating a set of similar software systems from a shared set of software assets. A common technique used to model the differences and commonalities (features) in software are Feature Models. Here, we use Feature Models to capture the commonalities and variabilities of a “Behavior Family” (a collection of similar behaviors). A concrete behavior is then specified as a configuration (a valid combination of features). In combination with a model-driven approach that allows for code generation, many variations of the same behaviors can be generated in this way.

Next to the normal top-down way of creating a software product Line (i.e. first make the feature model, then develop the different

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CGAT Conference 2010, April 6–7, 2010, Singapore.

Copyright 2010 CGAT

features), we also provide a bottom-up approach in which the feature model is extracted from the specification of individual behaviors, i.e. we provide a kind a reverse engineering of feature models. This last approach is provided because we don't want the company to completely change their way of working. This would require an investment that is too high.

Although the technique of SPL can be applied to all aspects of a game (i.e. also the static part, the game world), in this work, we only focus on the behavioral aspects of the computer game.

The structure of this paper is as follows. Section 2 describes the context in which the research has been carried out. Section 3 introduces the concept of Software Product Lines, Variability Modeling, and Feature Modeling. Section 4 presents the case study used to illustrate the approach. In section 5, the approach proposed is explained as well as an overview of the tool developed to support the approach. Section 6 relates our work to other work and finally, a conclusion and future work is given in section 7.

2. CONTEXT

The work presented here is an extension to the <*name*> approach, which is developed to support the development of behavior of interactive 3D/VR applications. The aim of the approach is to facilitate the construction of behavior in interactive 3D/VR applications by including an explicit *conceptual modeling* phase for behavior into the overall development process.

Conceptual modeling is the activity that creates technology independent models for the system to be constructed. During conceptual modeling, there is no need to consider implementation details; hence no technical background knowledge is needed to create these conceptual models. Therefore, these models can be used as a basis for discussing the design with different stakeholders. However, they can also be used as input for the implementation phase, and if they are precise and formal enough (like in our case) even code generation is possible.

The approach follows a model-driven design paradigm [15]. So-called Behavior Models (or Behavior Specifications) are expressed in a graphical way. These models are then translated into intermediate models. Ultimately, these intermediate models will be transformed into the actual implementation code. To obtain the final application, the code can either be directly loaded or has to be compiled first, depending on the output platform. Model-driven design elevates the specification of interactive 3D/VR applications to a higher level of abstraction than possible with low-level description formats or scripting languages. This makes it easier to create these kinds of applications but also to maintain them.

In combination with this model-driven approach, <name> uses generative design patterns [16], which allow a designer to specify behaviors by using patterns and customize them to suit a particular need. The design patterns are also specified at a conceptual level, thus they act as conceptual models and our approach allows generating code from them (hence the name *generative* design patterns).

The approach offers a number of advantages for the specification of behavior in computer games and other 3D/VR applications. On one hand, it allows reducing the size and complexity of the behavior specifications because the specifications are made at a conceptual level where we abstract from implementation details. On the other hand, the use of patterns provides a way to take profit of existing knowledge and experience, which may also contribute to a reduction of the development time and cost. In this paper, we will not further elaborate on these issues of the approach. We refer the reader to [18] and [19] for more information.

To support the <name> approach, an integrated development environment called <name> Studio (Figure 1**Fout! Verwijzingsbron niet gevonden.**) has been developed. This application enables a designer to create and maintain so-called Behavior Specification projects that are used to specify the behavior in a computer game. Based on these specifications, <name> Studio is able to generate a number of script files implementing the behavior of the computer game.

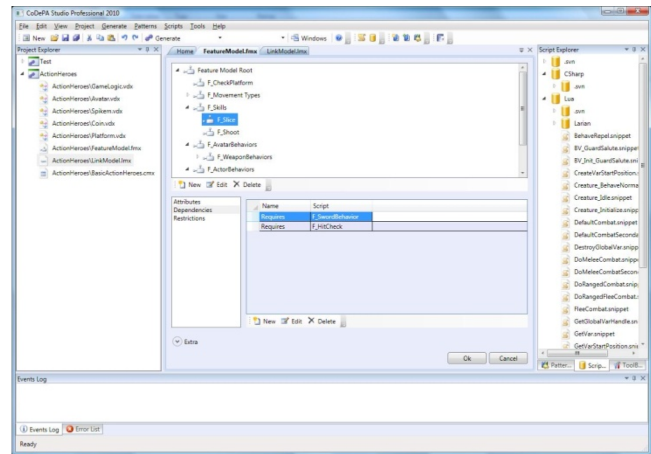

Figure 1. <name> Studio

An important part in our toolbox is the Conceptual Designer (Figure 2). This is a graphical diagram editor that allows creating the conceptual models in a graphical way. The tool has been implemented using Microsoft Visio [17]. A number of stencils are built containing the graphical representations of the different modeling concepts available in our approach. Examples of graphical elements can be found on the left side of **Fout! Verwijzingsbron niet gevonden.**. A detailed discussion of the different modeling concepts is beyond the scope of this paper. We refer the reader to [20] for more details on this. The graphical elements can be dragged from the stencils and dropped onto the canvas and proper connections can be made. Properties can be added, displayed and modified by the designer.

In addition, <name> Studio fully integrates the pattern oriented design approach mentioned earlier. It is possible for a behavior designer to easily search for a pattern in the existing collection of patterns. As a pattern is expressed by means of a graphical notation, the (generic) graphical representation of the selected pattern will automatically be dropped on the drawing canvas of the Conceptual Designer when selected. It is then possible for a behavior designer to customize the pattern to the particular context of use. It is also possible to construct new behavior patterns and add them to the existing collection of behavior patterns.

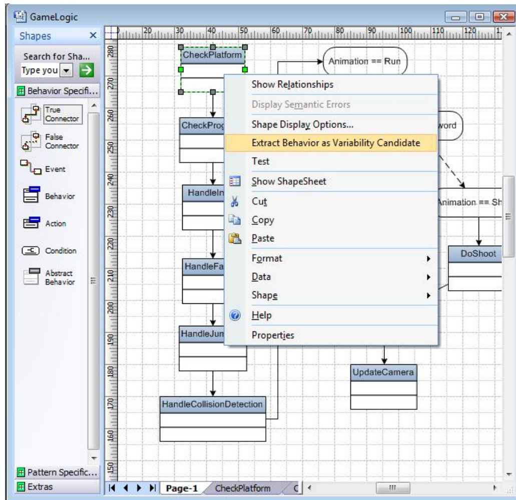

Figure 2. Conceptual Designer

3. VARIABILITY MODELING

Due to the large similarity in the software that companies delivered to different customers and/or for different platforms, the need for managing this variability arose. This has given rise to different variability modeling techniques as described in for example [8]. These techniques are typically used in Software Product Line Engineering [5]. A Software Product Line (SPL) is a family of similar software systems based on a shared set of software assets (also called features). Each software system (or product) is defined as a unique composition of features. Variability modeling then is the activity of expressing the common and variable parts, i.e. features, within the SPL and of defining the relationships and dependencies between these features. In this respect, variability modeling is also called feature modeling.

Table 1. Possible Feature Types

And indicates that all subfeatures must be part of any product of the product line	F1 F2 F2
Alternative indicates that only one subfeature can be selected in any product in the product line.	F1 F2 F2
Or indicates that one or more subfeatures can be selected as part of any product in the product line.	F1 F2 F2
Mandatory indicates that this subfeature is required as part of any product in the product line.	F1 F2
Optional indicates that this subfeature may or may not be part of a product in the product line.	F1 F2

SPL approaches usually follow a two or three-step process. The first step consists of the definition of the *Feature Model*. A feature model is a compact representation of all possible products of the SPL in terms of their features as well as their mutual relationships. A feature model is mainly a hierarchical structure (i.e. decomposition) of features. Commonly, there are five types of decomposition relations possible in a feature model [14]; Table 1 shows their graphical notation and meaning. In addition to these feature types, which define feature relations based on their composition, additional constraints (or dependencies) between features may exist. Constraints describe how features depend on each other, e.g., the use of one feature may require and/or exclude the use of another feature. Figure 3 shows a sample feature diagram for the Order Process Problem introduced in [24]. The feature model shows the order process's composing features (Basket, Fulfillment, and Transaction, which are on their turn decomposed) and how they are related based on the notation given in Table 1. A feature's contribution to variability is given via its feature type. A feature that contributes to variability is called a *variable feature*. In the above example, *Shipping_Cost* and *Invoice* are variable features because *Shipping_Cost* is an optional feature and for *Invoice* there is the option to use a *Printed_Invoice* or an *Online_Display* or both

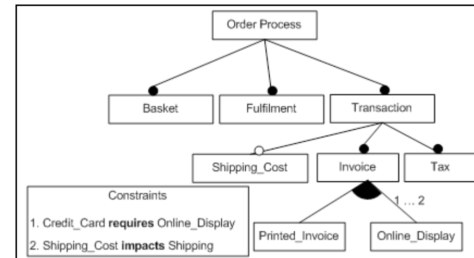

Figure 3. Example Feature Diagram

Accompanied with additional feature dependency constraints, a feature model gives information about the features that should be part of a valid software product. For instance, in the example, the use of the feature *Credit_Card* requires the use of the feature *Online_Display* (given by the constraints expressed in the text box at the left side of Figure 3).

To specify a specific product, the next step is the specification of a *Configuration Model*. A configuration model is a representation of a valid composition of features and is created by declaratively selecting or deselecting features from the Feature Model according to the product's needs. A valid composition is a composition that meets all the type restrictions and feature dependencies imposed by the feature model. A valid composition results in a valid software product. Such a particular configuration, or software product, is also called an instance of the SPL.

To come to an actual implementation for a configuration, the features have to be realized, i.e. implemented by means of a set of implementation artifacts (e.g., classes, methods, packages, …). Since the feature models are typically used during design and the implementation artifacts are created during implementation, some sort of mapping is needed between features and the implementation artifacts in order to link the two levels and to enable the automatic generation of SPL instances. This mapping is defined in what is typically called a *Component Model*. A Component Model describes the internal structure of the individual components, their parts as well as references to the actual artifacts. A component is an abstract element which corresponds to a possible variation point in the implementation. It is further defined by at least one part. A part is a concrete element which is further defined by an actual artifact. Constraints describe how the parts are bound to features (i.e. through the *hasFeature* constraint).

Once these three models are specified, and the implementation artifacts are created, the remaining part in the generation process can be performed automatically [9]. Based on the Configuration

Model, defined as an instance of the Feature Model, the different parts (i.e. implementation artifacts) can be selected, using the mapping defined in the Component Model. This process is displayed on Figure 4.

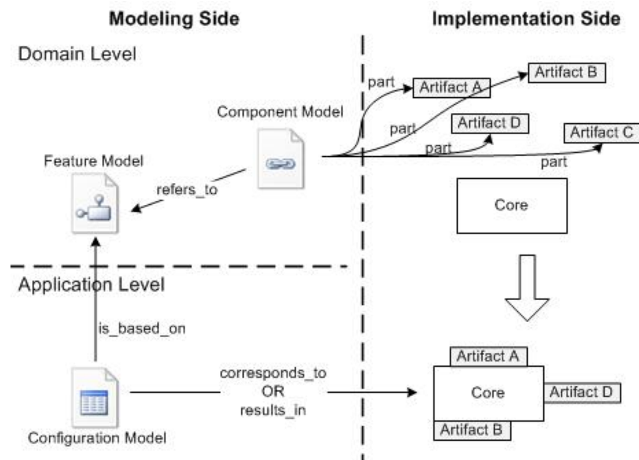

Figure 4. Feature Modeling Transformation Process

In this paper, we will use feature models to deal with variability in behavior, i.e. to describe the variations of a behavior. Note, that currently we do not use feature models and SPLs as a mechanism to specify an entire "game family".

4. CASE STUDY

To illustrate our approach, an example game is used, which is a small online-based computer game. This game is called "Action Heroes" and is written in JavaScript. In addition, it uses the Open3D (O3D) engine and its API [10] to deal with rendering the graphics. O3D is an open-source web API/engine which can be used to create rich, interactive 3D applications in the browser. The graphics themselves are made with Google SketchUp [11]. Figure 5 presents a screenshot of this game.

In the game, the player is controlling a little warrior. The mission of this warrior is to reach the end of the level without getting hurt and collect as many coins as possible in the process. In order to make the game attractive, a number of static and moving pads need to be used to cross some hurdles. The warrior has both a sword that he can use as well as a crossbow with which he can shoot arrows. The coins are typically rotating and can be picked up by colliding with them. During the course of action, the warrior is being hindered by so called spikems, yellow balls covered with sharp spikes. These spikems can either spin or charge, or both.

Although this is a small game, already some variability can be recognized that may also be useful in similar games. Typically, most avatars will move but there can be a difference in their movement style. That is, the hero avatar can walk, run, jump, fly, and so on. Different games might also use different algorithms for executing these actions. In addition, a number of different types of weapons can be used by an avatar and each weapon will be associated with a different action that can be performed, for example a sword can be used to slice; a crossbow can be used to shoot. Different animations could be chosen to visualize these actions. Also, usually, the non-player actors can also perform a number of behaviors, such as moving and rotating. And, depending on which weapon(s) is (are) chosen, a number of "check functions" need to be performed, for example to check when an actor is hit or sliced.

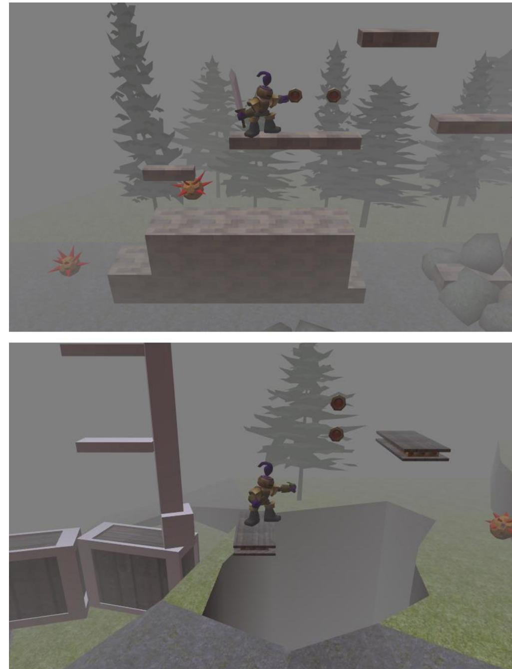

Figure 5. Screenshots "Action Heroes" game

5. SUPPORTING VARIABILITY IN BEHAVIOR MODELING

5.1 Approach

In this section, we explain how the <name> approach and associated software tool has been extended with supporting capabilities for variability by using the concepts of SPL.

In our approach we provide support for variability in two ways. In the top-down approach, a Behavior Family (i.e. a SPL for a behavior specification) is being built from scratch by analyzing the domain under consideration, trying to identify possible commonalities and variabilities, and translating this into a proper feature model. Then, based on the feature model, a behavior specification (i.e. a conceptual model) can be developed for the Behavior Family. In the reactive or bottom-up approach, the Behavior Family is being built up from extracting the commonalities and variabilities from a set of existing behavior specifications and translating this into an initial feature model. Next, this initial feature model can be adapted or extended to allow for more variants. While the first approach is appropriate when someone wants to start from scratch, the second approach is more suitable for companies with existing behavior specifications where variability was not considered when designing the behaviors. In this paper, we will focus on the bottom-up approach. This approach is also suitable to incorporate new developments from the fast evolving computer gaming domain into existing specifications, e.g., to introduce variants for new interaction devices.

The starting point for this bottom-up approach is a single behavior specification. The goal is to generalize this behavior into a Behavior Family that can be used later on to easily generate a

number of variants (or different flavors) of the behavior (among which the original behavior we started from). The principle is as follows: From the original behavior specification we identify possible sub behaviors that can be variable, i.e. which are specific for this particular behavior but could be different in other situations. These are candidate variation points in the feature model to be created. Next, such a behavior is replaced in the behavior specification by a *variation point behavior* and a feature tree is constructed that reflects this variability. Initially the feature tree consists of a feature with one mandatory subfeature. In later steps, more "alternative" or "or" subfeatures can be added for this feature to introduce the variability. In addition, also a component models is constructed that maps these features onto the correct behavior specifications.

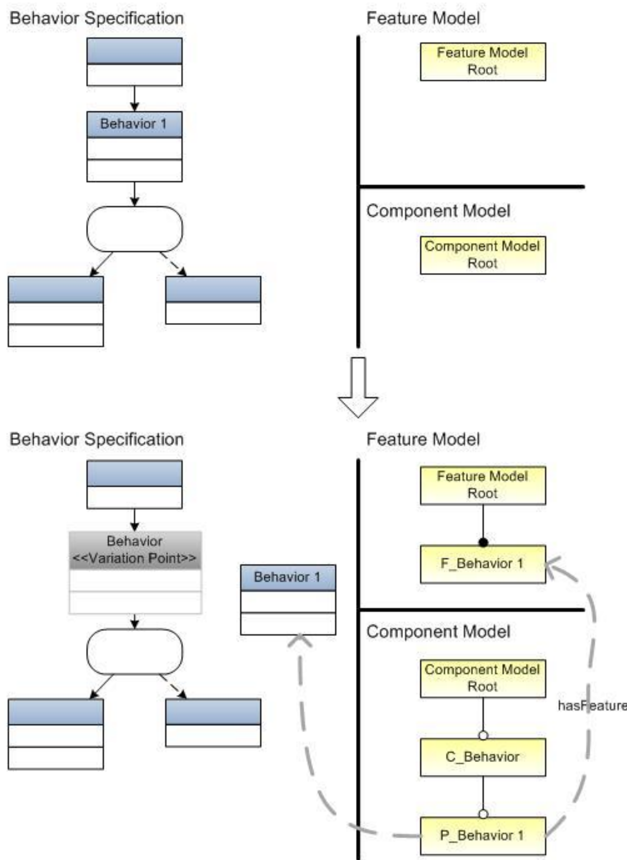

Figure 6. Extraction Process

Figure 6 depicts this process. The top half of Figure 6 displays the initial situation, an existing behavior specification (left) with a number of actions (a rectangular box divided in two), behaviors (a rectangular box divided into three parts) and conditions (a rounded rectangle) as well as a feature model containing no features (just a root) (right above) and a component model containing no components/parts (also just a root) (right below). The behavior "Behavior 1" has been identified as a possible candidate for being variable; hence it is extracted from the overall behavior specification and replaced by a variation point behavior. The bottom half displays the result of this process. Behavior 1 is now a stand-alone behavior or a *variant behavior*. In the overall behavior specification, "Behavior 1" is replaced by a variation point behavior "Behavior". It is a kind of abstract behavior and serves as a placeholder for the actual behavior that will be used when a configuration is created. This element is represented in a similar way as a regular behavior, i.e. a rectangular box, divided into three parts, but is color is grey and the top compartment has the text "«Variation Point»".

The newly created feature model is shown in the lower part of Figure 6. This model consists (currently) of a single mandatory feature, called "F_Behavior 1". The feature "F_Behavior 1" represents the variant behavior "Behavior 1". Therefore, a link is needed between the feature and the actual behavior specification. This is done through a component in the component model. The newly created component model is also shown in the lower part of Figure 6. Note that for both models the notation from the Feature Modeling DSL, a plugin for Visual Studio, is used. It shows that a new component has been created, called "C_Behavior" implicitly referring to the variation point behavior "Behavior". It is consisting of a part "P_Behavior 1" that corresponds to the feature "F_Behavior 1" through the *hasFeature* relationship. This part also refers to the behavior "Behavior 1" in the behavior specification.

Both the component model and the feature model are very simple at this moment since we just extracted a single behavior. Actually, no variability is possible at this moment. However, as we iterate the process, extracting more behaviors, the models will grow and evolve over time. The developer can also explicitly add new behaviors (and respectively features and components), for example to define another variant of an already extracted behavior. It is also possible to apply this extraction process on extracted (isolated) behaviors themselves, i.e. variable subbehaviors can be extracted from such an extracted behavior as well, giving rise to a sub tree of an existing feature in the feature tree and subcomponents of existing components in the component model. Note, that since we are supporting the top-down approach as well, both the feature model and the component model can be further refined after the extraction process has been finished. At that moment, any (additional) dependencies can be added.

Once the feature model has been created completely, either from scratch or through the extraction process, a concrete behavior can be specified by means of a configuration model representing a concrete instance of the behavior family. The configuration model shows the features selected from the feature model (not shown here).

5.2 Example

In order to illustrate the approach, we show how we can create a behavior family from the behavior specification for the "Action Heroes" game introduced in section 4. As this is a simple game, we only have a few behavior specifications. This is used as the starting point. First, we should identify and extract the subbehaviors that could be variable. This can be done through our <name> Conceptual Designer by using the menu-option "Extract Behavior as Variability Candidate". This starts the extraction process, which is presented to the user by means of a wizard interface, which guides the user through the process.

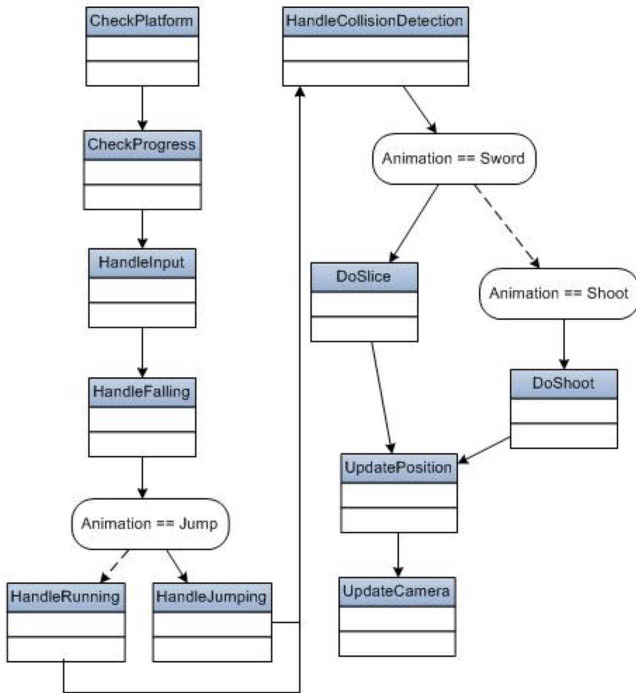

Figure 7. "GameLogic" Specification Extract

The main behavior is the "GameLogic", which represents the actual game loop. Figure 7 displays a simplified extract from its behavior specification. This behavior begins with checking the platform (only when the avatar is able to jump). Then, the overall progress of the avatar in the game world is checked. This needs to be executed in every game of this kind. When this is done, the input from the user is handled. Depending on the keys that the user has pressed, a certain animation (for the different movement types or skills) is being set. Next, some handling functions are executed for dealing with the movement such as falling and jumping. Obviously, these only need to be executed when the avatar is able to jump and/or to fall. Afterwards, the collision detection is handled which is always executed. This is followed by the execution of the actual skill animation set by the handling of the input. Finally, the correct pose of the avatar is updated as well as that of the camera. The behaviors for the other objects (i.e. spikems, coins and platforms) are dealing with a particular standard movement or playing a particular sound. Also there is some subbehavior that in every loop of the game checks whether it is hit or not.

The process of turning this behavior specification into a behavior family happens as follows. Starting with the "CheckPlatform" behavior, we know that this only needs to be executed when the avatar is able to jump. So, this behavior will not be necessary in all games of this kind (i.e. not in games where jumping is not allowed). Therefore, it is extracted and mapped onto a feature called "F_CheckPlatform". The CheckProgress and HandleInput behaviors is needed in all games and therefore can remain untouched. Next, the following behaviors in the sequence deal with the movement. Here, we will again perform some extractions. For example, the "HandleFalling" behavior can be extracted and replace by a more general type of movement handling behavior. We will now show into more details how this extraction is done.

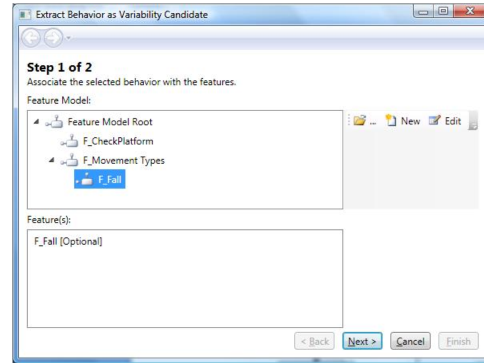

Figure 8. Step 1: Feature Model

In the first step, a tree-structure is presented showing the current feature model (see Figure 8) in the project. Here, the designer should associate the selected behavior with one or more features. There is also the option to add new features to the feature model. In this way, he can already add new subfeatures to a feature. A new feature is added by giving it a name and indicating its variability type (i.e. And, Mandatory, Optional, Or, Alternative).

In our example, we have inserted a new feature called "F_Movement Type" with a subfeature called "F_Fall". This is an optional feature, meaning that it is not required for the designer to select this feature in an actual configuration. At this moment, the feature is not yet linked to the actual behavior. This is specified in the second step.

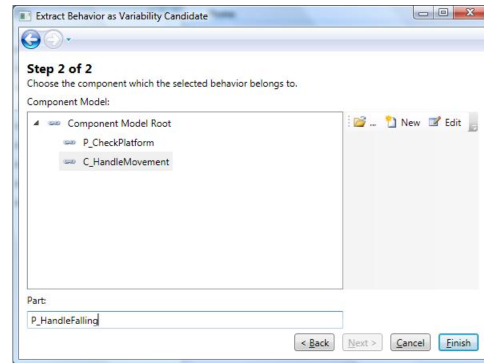

Figure 9. Step 2: Component Model

In the second step, the user is presented with a tree-structure showing the current component model (see Figure 9) in the project. Here, he needs to associate the feature (from step 1) with some component. Next, the name of the part itself has to be given. At this stage, also an option is provided to create a new component and insert it directly into the component model. Adding a new component will be reflected in the behavior specification by replacing the behavior by a variation point behavior.

In this example, we have created a new component called "C_HandleMovement". The part that is introduced is called "P_HandleFalling". This part is now linked to the selected feature "F_Fall" from the first step. The part will also refer to the extracted behavior, in this case "HandleFalling".

Various other movement type features can now be added as well (e.g., for HandleRunning and for HandleRunning) by extracting those behaviors, creating subfeatures of F_Movement Types in the feature model and by creating parts below the

C_HandleMovement in the component model. Once the designer has completed the two steps of the wizard, the <name> Studio makes sure that the necessary changes are made on the behavior specification (i.e. replacing the behavior with a variant point behavior) and that the correct updates are done in both the feature model and the component model (i.e. features and components/parts are added and the links are created).

Next, the "HandleCollisionDetection" behavior is also required in every game and therefore can remain. However, the following behaviors in the GameLogic sequence deal with the different skills that an avatar has (or can have). The same approach as for the movement types can be followed here. Following this same process all the way through the behavior specification allows us to come to our behavior family.

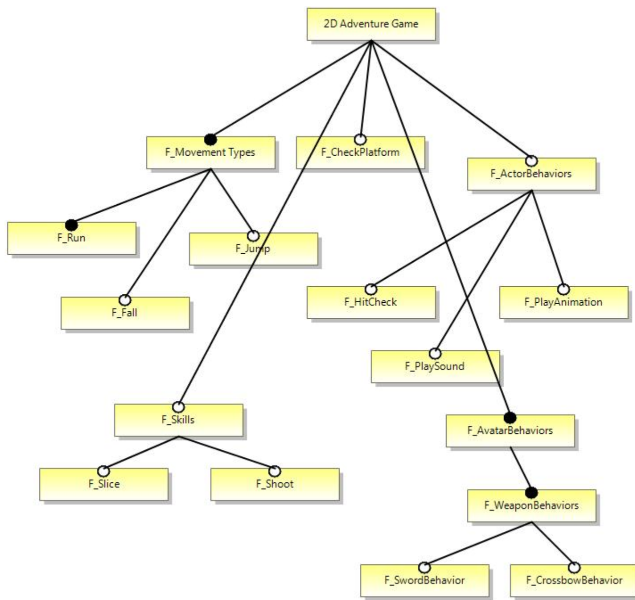

Figure 10. Feature Model of a 2D Adventure Game's Behavior

The resulting feature model is given in Figure 10. The feature model has a F_CheckPlatform feature to check whether or not an avatar is positioned on one of the moving platforms. The F_Movement Types feature contains a number of subfeatures, each providing a different movement for the avatar. In the same way, the F_Skills feature provides various skill features that an avatar may have. The F_AvatarBehaviors feature contains a F_WeaponBehavior feature. The F_WeaponBehavior feature can either be a F_SwordBehavior or a F_CrossbowBehavior. Finally, there is the F_ActorBehaviors feature that is further specified as having a F_PlayAnimation, a F_PlaySound and a F_HitCheck feature.

Several constraints exist between the features. For example, if the F_Slice feature is selected, the avatar will need to have the F_SwordBehavior. The same accounts for the F_Shoot and F_CrossbowBehavior. Furthermore, if the F_Jump feature is selected, the F_CheckPlatform needs to be there as well.

Note that editing and even creating a new behavior family is also possible using the dedicated model editors within the <name> Studio. Inside the tool, the models can be opened (see Figure 1), new elements can be added, and existing elements can be deleted. Furthermore, from each of the elements, the attributes, the dependencies as well as the restrictions can be edited. This allows, in a later phase of the development process, to improve the models created using the top-down approach

6. RELATED WORK

The problem of variability in software and product line architectures have been discussed in many papers [2][19]. More specifically, the concept of feature model was first introduced by Kang et al. [14] in the Feature Oriented Domain Analysis (FODA) method, to help the identification of important or domain specific properties during the analysis phase. Others notations have also been proposed to expand the feature model representativeness and to provide support to different types of structural relationships. For example, Czarnecki and Eisenecker [7] use XOR and OR relationships to represent alternative and mutually exclusive features, and in [6], they propose a feature cardinality-based notation. Others have used ontology to represent and verify features. More details on this type of work can be found in [23] and in [21].

There is not a lot of research done on feature models for Game development. We came across two major researches. The first was done by Zhang and Jarzabek [25]. They proposed a Role-Playing Game (RPG) product line architecture (RPG-PLA) for mobile phone. In their approach, they capture and group similarities as well as differences among four RPGs. As from there, they can develop a RPG feature model. Although, they use feature diagram for doing this, they only did it from an architecture point of view i.e. according to the type of mobile phone and display resolution and platform. They did not use feature diagrams on the aesthetics part of the game itself. Our approach is different as it focuses more on the aesthetics of the game and more specifically on the behaviors.

The second major work was done by Alves [1]. He presented a method for managing a SPL for mobile games. Like Zhang and Jarzabek [25], his approach also uses feature models. His approach focuses more on how to manage a SPL and it is also oriented towards the architecture being used (platform, mobile phones) and less on the aesthetics of the game.

7. CONCLUSIONS AND FUTURE WORK

In this paper, an approach to support variability in behavior specifications has been presented. The approach is an extension to the existing <name> approach that aims to facilitate the authoring of behavior in computer games and other interactive 3D applications. The approach utilizes techniques from Software Product Line engineering.

We have described how software variability techniques can be used to cope with different flavors of behaviors. A feature model is used to describe the commonalities and variabilities for a set of similar behaviors. A component model is used to describe the link between the feature model and the actual artifacts (small behavior specifications). We also have described an approach to extract a feature model from existing behavior specifications. This bottom-up approach does not require creating a feature model from scratch, which can be difficult for a developer because the required variability may not yet be clear when one starts to design a game. In addition, this bottom-up approach is also more suitable when different similar behaviors already exist. The feature model and component model are gradually extracted and constructed. A configuration model can then be used to specify a particular instance (flavor) of the behavior. The approach is supported by an

authoring tool that supports the extraction process as well as the editing and creating variability models. In addition, code can be automatically generated from the behavior specifications. Currently, we generate LuaScript and JavaScript.

Other work which has been performed in the area of variability modeling and computer games also address features such as screen size, platform, memory, computer power, connectivity and so on. In this work, we have only focused on behavioral features. Nonetheless, it would also be interesting to apply the approach on these kinds of features.

8. ACKNOWLEDGMENTS

This research is carried out in the context of the <name> project (<project>) which is directly funded by the <organization>. The research is performed in close cooperation with <company>, a <country> game developing company.

9. REFERENCES

[1] Alves, V., Matos, P.Jr., Cole, L., Borba, P. and Ramalho, G. 2005. Extracting and Evolving Mobile Games Product Lines. In 9th International Software Product Line Conference (SPLC-EUROPE 2005), Rennes, France.

[2] Bosch, J., 2002. Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization. In: Proceedings of the Second Conference Software Product Line Conference (SPLC2), pp. 257-271.

[3] Bosch, J. 2000. Design and Use of Software Architectures – Adopting and Evolving a Product Line Approach, Addison-Welsey

[4] Busby, J., Parrish, Z., VanEenwyk, J. 2004 Mastering Unreal Technology: The Art of Level Design. Sams Publishing.

[5] Clements, P., Northrop, L. 2001. Software Product Lines: Practices and Patterns, Addison-Wesley.

[6] Czarnecki, K., Kim, C.H.P. 2005. Cardinality-Based Feature Modeling and Constraints: A Progress Report. In: OOPSLA'05 International Workshop on Software Factories

[7] Czarnecki, K., Eisenecker, U.W. 2000. Generative Programming: Methods, Tools, and Applications. Addison Wesley

[8] Deelstra, S., Sinnema, M. and Bosch, J. 2004. Experiences in Software Product Families: Problems and Issues during Product Derivation, Proceedings of SPLC2004, Boston, Aug. 2004, LNCS3154, Springer-Verlag, pp. 165-182

[9] Dollard,K., 2004. Code Generation in Microsoft .NET : Apress.

[10] Google, 2009. Open3D Application Programming Interface. [Online: http://code.google.com/apis/o3d]

[11] Google, 2009. Google Sketchup Reference Guide. [Online: http://sketchup.google.com]

[12] Ierusalimschy, R. 2003. Programming in Lua. Lua.Org. 1 edition.

[13] Instituut Samenleving & Technology (IST). 2009. Jaarmagazine 2009.

[14] Kang, K., Cohen, S., Hess, J., Novak, W. and Peterson, S. 1990. Feature-Oriented Domain Analysis (FODA): feasibility Study. CMU/SEI-90-TR-21, SEI, USA.

[15] Kleppe, A., Warmer, J., Bast, W. 2003. MDA Explained: The Model Driven Architecture(TM): Practice and Promise, Addison-Wesley.

[16] MacDonald, S., Szafron, D., Schaefier, J., Anvik, J., Bromling, S. and Tan, K. 2002. Generative design patterns. In Proceedings of the 17th IEEE International Conference on Automated software engineering. Edinburgh, Scotland. pp. 23-33.

[17] Parker, D.J. 2007. Visualizing Information with Microsoft Office Visio 2007, McGraw-Hill Osborne Media.

[18] xxx

[19] xxx

[20] xxx

[21] Peng, X., Zhao, W., Xue, Y., Wu, Y. 2006. Ontology-Based Feature Modeling and Application-Oriented Tailoring. In: ICSR 2006, pp. 87-100

[22] Thiel S. and Hein A. 2002. Systematic Integration of Variability into Product Line Architecture Design. Proceedings of SPLC2002; LNCS 2379, San Diego, California, pp. 130-153

[23] Wang, H., Li, Y., Sun, J., Zhang, H., Pan, J. 2005. A semantic web approach to feature modeling and verification. In: Workshop on Semantic Web Enabled Software Engineering (SWESE'05)

[24] Ye, H., Liu, H. 2005. Approach to modelling feature variability and dependencies in software product lines. In: *Software, IEEE Proceedings - Volume 152, Issue 3*, Page(s): 101 – 109

[25] Zhang, W. and Jarzabek, S. 2005. Reuse without Compromising Performance: Experience from RPG Software Product Line for Mobile Devices. In:Proceedings of the 9th Int. Software Product Line Conf., SPLC'05, Sept. 2005, Rennes, France, pp. 57-69.