Artificial Intelligence in the Game Design Process 2: Papers from the 2013 AIIDE Workshop (WS-13-20)

A Declarative Domain Model Can Serve as Design Document*

David Llansó, Pedro Pablo Gómez-Martín,
Marco Antonio Gómez-Martín and Pedro A. González-Calero
Dep. Ingeniería del Software e Inteligencia Artificial Universidad Complutense de Madrid, Spain
{llanso,pedrop,marcoa,pedro}@fdi.ucm.es

Abstract

Detailed design documents have been criticized as a hard to maintain artifacts that may easily become useless while a game under development keeps evolving. In this paper we propose the use of declarative domain modelling as a communication tool and a form of contract between designers and programmers. We show how this model, including entities and actions relevant for the game design, can also serve to support debugging tools for game designers.

Introduction

Videogames are large and complex real-time applications that must cope with an enormous amount of multimedia assets, while, at the same time, providing an spectacular and immersive experience, and, most important, being fun to play.

Since the secret recipe for fun in games is revealing elusive and hard to find, game development is turning into a trial-and-error design process. As the budgets increase and in order to reduce the risk of shipping an expensive to develop but unsuccessful game, development teams start to playtest their games as soon as a playable prototype is available (Swift, Wolpaw, and Barnett 2008). Thus, game design can steer according to feedback from actual players. This process goes from early pre-production stages, in order to validate a game idea, up to late production stages in order to balance and fine-tune the game mechanics.

Building the dynamics in a game is a collaborative effort between level designers and programmers. Programmers provide to the designers with the *building blocks* for specifying behaviour in the game, as a collection of parametrized systems, entity types and actions those entities may execute. Designers use combination of state machines, scripting, visual languages and map editors to build complex behaviours by composing the basic pieces the programmers provide.

Ideally, a detailed design document should serve as the specification contract between designers and programmers: before entering into production stage, it should be perfectly clear which building blocks the programmers should build and what building blocks the designers would count on for designing the game levels. However, in actual development, the design of the game usually becomes a moving target, with designers coming up with new requirements for programmers as new mechanics are explored. Furthermore, programmers overwhelmed by their current tasks can feel tempted to let designers use their dubious scripting skills to implement such additions, what, later on, will probably result in the programmer debugging a designer's script during crunch time.

In this paper we propose the use of declarative domain modelling as a communication tool and a form of contract between designers and programmers. We show how this model, including entities and actions relevant for the game design, can also serve to support debugging tools for game designers and thus provided a more controlled environment for behaviour specified by designers. Next Section elaborates on the problems of detailed design documents and motivates the benefits of explicit domain modelling as design artifact. The method section describes a methodology for game development built using a domain model as the main interface between programmers and designers, while the example section exemplifies this methodology with a simple development. Finally, the last section presents some conclusions and future work.

Problems

Over the years videogames studios and professionals have evolved, adapting their internal process to the new realities. However, there are still many issues that have not been solved.

Nowadays there is a big dilemma regarding the design document. There is no accordance about its importance and the details it should contain about the features the final videogame must incorporate. It is broadly recognized that, mainly because of its structure and extension, the document is usually forgotten by designers and developers in the final stages of the development (Keith 2010). The main reason is the volatile nature of the videogame itself, that even in these final stages of the process have designers introducing important changes on its design (Salen and Zimmerman 2003). These late changes on the death throes of the development are usually not added to the document due to time pressure and productivity. So, though it is important to have

*Supported by the Spanish Ministry of Science and Education (TIN2009-13692-C03-03)
Copyright © 2013, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

a central document used by developers which contains the designers ideas, the typical design document used by most of the studios are considered hard to maintain up to date and inefficient (Dormans 2012).

A stale design document forces the communication between designers and programmers to be oral. Though this is not inherently bad, the fact that these conversations do not have a match in any permanent document is not desirable. Moreover, the existence of this document would not guarantee the understanding between them, due to the inherently ambiguous nature of the natural language. Therefore both academy and industry have taken some effort to solve the issue. Different authors propose the convenience of having a language or an unified design method (Zagal et al. 2005; Zagal and Bruckman 2008) that could be used to create unambiguous descriptions of current and new games to completely remove the misunderstanding between developers and designers. Other authors advocate for using the description of the central elements of the game such as the possible decisions the player can make, the set of actions it can perform or the abilities it may gain over the game (Salen and Zimmerman 2003; Cook 2007).

Regarding the formal description of the game design, there are several contributions that completely remove the game design document or at least complement it with some kind of declarative representation. Some efforts have been made in incorporating semantic information to game elements (3D models) (Tutenel et al. 2011; Zhang et al. 2012; Gaildrat 2009). With this technique, the game domain is expressed through formal descriptions of the entities that conform them and this information may be used not only as a consensus between designers and programmers but also to procedurally create game levels or at least to check the consistency of the hand-made ones. A different way of the description of this formal model is the use of *Answer Set Programming* were the team creates a set of rules using logic programming that are later used for the procedural content generation (Smith and Mateas 2011; Smith et al. 2012). Though both approaches have some advantages (such as a better automatization of certain processes), its acceptance in the game studios is low (or inexistent).

A different approach to describe the game domain that may be found in the academy is the use of ontologies. Some efforts have already been made to analyze existing games and build up an ontology with the set of concepts that appear on them (Zagal et al. 2005; Zagal and Bruckman 2008). One of the advantages of using an accepted ontology to described new games is the increase of the understanding. However, ontologies can contribute during the game play: the development team can describe the elements of a videogame in the ontology and then use them to infer new knowledge (Machado, Amaral, and Clua 2009).

Going a bit further, some designers and researchers found alternative methods that allow designers themselves to create prototypes without the needed of programmers (Neil 2012). Ideally these tools should have as input the formal description of the game domain and as output the desired prototype. Unfortunatelly, building such a tool is quite hard if not impossible. Current tools are easy-to-use applications that allow designers and other users without programming knowledge the creation of these small prototypes using a limited set of features. But as the main goal of these tools is to make easy the creation of prototypes, they provide with simple and easy-to-use interfaces that, as a side effect, limit the expressibility and creativity of their users (Neil 2012). Therefore these tools are not useful in a professional game development because designers cannot try novel game design ideas (Nelson and Mateas 2009).

As a conclusion, it seems that nowadays is not still possible to completely remove the role of programmers in the prototype development. Current modelling tools (Sicart 2008; Araújo and Roque 2009) allow designers to visually define the semantic model of the game and, in some cases, the procedural generation of content, but no more. Therefore, being programmers still needed, we should try to improve the communication between them and game designers.

Method

Experience shows that game design must be accomplished not only by game designers but also by the programming team that will eventually develop the game. After all, they will be in charge of overcome the technical challenges the game imposes, and know where the platform limit is (Long 2003) so they must restrict the designer ideas when they become too ambitious.

This fact makes clear the necessity of a good communication between both teams, designers and programmers. Our proposal consists of the use of a *formal domain* of the videogame that define the entities the game will have and their abilities. For example, the domain will specify that a specific kind of entity can attack, and other one can run. This formal description becomes a pivotal point in the game development, and constitutes the reference point where designers and programmers look for information.

In that sense, we have developed a visual authoring tool called *Rosette* (Llansó et al. 2011a) that fills the gap between programmers and designers and serves as a contract between them. The tool is aware of these two different roles and presents different information and options depending on the role of the current user. Figure 1 shows a general view of the architecture.

The formal domain created in *Rosette* is not written in stone: it can be changed and adapted when the game evolved. This is also true for design documents but, as described above, they are hardly updated once the development process has begun. Fortunately, having a formal domain description in *Rosette* provides enough advantages to convince designers and programmers to keep it up to date. Specifically it not just automates some code generation and refactorization (good for programmers), but also makes consistence checking in the created game levels (good for designers). In this way, *Rosette* promotes and eases agile development methods based on incremental and iterative development. It can also monitor level creation and perform consistence checking to avoid deviations from the game model that could create bugs hard to find in the long run.

The overall process of each development cycle consists of the next steps:

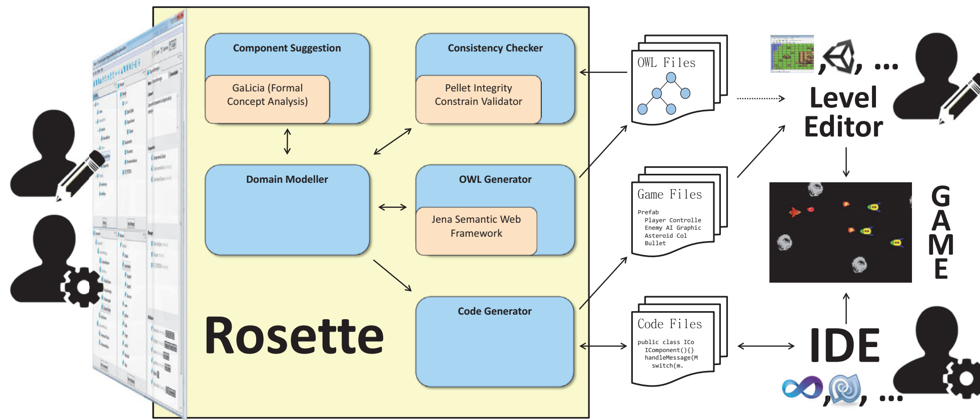

Figure 1: General view of the *Rosette* architecture

- Together with the classic design document, designers formally define (or change) the game entities and their abilities using an easy-to-use graphical interface (Llansó et al. 2011a).
- Programmers analyze, negotiate and validate the new game domain, that becomes a *contract* between designers and programmers.
- Programmers enrich the initial domain adding "technical entities" that have no semantic importance for the gameplay and designers could have left out. They also identify the *software components* for all those entity types (Llansó et al. 2011c; 2011b) and *Rosette* will generate a lot of scaffolding code for the game. It also generates metainformation for the game level editors referring the available entity types.
- Programmers "fill in the blanks" *Rosette* has left in the generated code, developing the game specific functionalities in the software components.
- Designers create game levels and also define game logic using the behaviour tool chosen in the project, such as Finite State Machines (FSMs) or behaviour trees (BTs).

Currently *Rosette* is able to generate scaffolding code for Unity[1] and for a private C++ game engine used by the authors. It also generates information for the game level editors by means of *prefabs* (Unity) or blueprints and archetypes files (private engine).

Although *Rosette* can be easily extended to generate code for any other platform and language, in this paper we will focus on Unity, and the aids that it can provide to designers while create the game levels using its built-in IDE.

While defining a new level map, designers are in charge of three main tasks:

- Create the *environment* of the map, using static meshes such as walls, corridors, terrains, trees, buildings or whatever the game needs.
- Add interactivity by means of *entities*. *Rosette* would have generated *prefabs* using the domain model with the accepted entities, containing those software components that programmers have approved. Some of the components would be *scripts* that could be not developed yet, but programmers will be eventually fill them in.
- Define entities behaviours using Finite State Machines (FSM) or any other friendly AI technology, and incorporate them into the entities using a special component designed to run them.

[1]http://unity3d.com/

It is important to note that the formal model specified in *Rosette* contains information about entities referring their abilities (actions they must perform such as "*Attack*" or "*MoveTo*"), senses (perception capacities such as "*Enemy-Hidden*" or "*SufferDamage*") and state ("*health*" or "*armor*"). Nevertheless, the formal domain *does not* contains *when* an entity attacks or what it would do when *SufferDamage* because that would depend on the concrete map and difficulty level.

Entities become therefore the game level *building blocks*, and designers must create their concrete behaviours using a FSM editor. *Rosette* is used in this phase to *validate* the FSMs testing if all the actions used in the FSM states are available for the entity, and whether all the events specified in the transitions can be "sensed".

On the other hand, designers could be tempted to create their own entities joining together the *components* available instead of just using the prebuilt (and approved) entities (*prefabs*). Playing around with new entities is not necessarily bad, and provides a way for designers to experiment and test new ideas without affect the formal domain. Unfortunately, the component dependencies could be unattended and the resulting entities could suffer of inconsistencies because, after all, some of the components are artifacts useful for programmers that designers usually ignore. *Rosette* can be used also in this moment to look for inconsistencies in the map, and will analyse the new entity types in search of problems. It should be notice that once the new entities have been tested and designers consider that they are useful, they should be incorporated back into the formal domain so they will become *prefabs* in the next game iteration. Inclusion of new entities could result in code refactorization in order

to reuse or split previously defined components, so entities created manually should be approved by both designers and programmers with the aid of *Rosette*.

In fact, although the assistance provided by *Rosette* is useful from the very beginning, where our tool really shines is in subsequent iterations. When the game domain is changed, *Rosette* is able to track the differences between the old and new versions and refactor all the code preserving that one added by programmers to the generated templates, and moving it around to their new locations when needed. This constitutes an unvaluable tool for the agile development cycle because the error-prone task of changing code is avoided.

On the other hand, designers also loose the fear to changes because the old entities (prefabs) are automatically updated with the new ones, and parameters specified in the Unity IDE (such as the initial energy of an entity) is automatically updated even if that value is moved from a component to another one. FSMs are also reevaluated, so all the tests that they suffer while being created are executed within the new domain in order to discover inconsistencies such as an action that has vanished and a FSM is still using.

All this consistence checking (similar in spirit to *test-driven development*) for both programmers and designers makes viable an agile development cycle and is in some sense immune to changes and additions in the game design.

Under the hood, the complete game domain in *Rosette* is stored using OWL[2], a knowledge-rich representation that allows inferences and consistence checking. Entity consistency checking (Llansó et al. 2011a) can be done using the *Pellet Integrity Constraint Validator* (Pellet ICV)[3]. Pellet ICV extends the Pellet reasoner (Parsia and Sirin 2004) by interpreting OWL axioms with integrity constraint semantics. That means that Pellet ICV interprets OWL ontologies with the Closed World Assumption in order to detect constraint violations in RDF data. Pellet ICV also provides automatic explanations of why integrity constraints are violated, so we use them to provide the correct feedback to the user.

Rosette uses a component-based representation for managing the entities in a game (West 2006) so the OWL entities are described by components (pieces of functionality), which encompass messages (that represents actions and senses) and attributes (simple or complex data types with some constraints). When entity prototypes are defined in *Rosette* it creates an OWL domain representation that is shared in all the levels of the game and it is considered the *static model*. However, when creating a level, designers create new entities, change the default attribute values and create some AIs. All these information is also integrated in our OWL representation to check the *level consistency* and constitutes the *dynamic model*. The concrete OWL representation is broadly explained in (Llansó et al. 2011a) and, in the example section, there are examples of all these concepts.

Example

In order to validate and show our method we have developed a small game example using Unity and *Rosette*. This example consists of a simple 3D action game where the player controls an avatar and the system controls several non-player character (NPC), which can do different things such as protect an area, explore the environment or look for the player. Designers are responsible for defining the game in depth, determining the details of the playability, aesthetic aspects, history, etc. According to our methodology, this means that they are in charge of creating a hierarchical distribution of entity/game object prototypes in *Rosette*. These entity prototypes are defined with the state and functionality in terms of attributes they have and actions they are able to carry out. Designers can also set values to the attributes that populated the entity prototypes although these values will be editable later in a per instance basis.

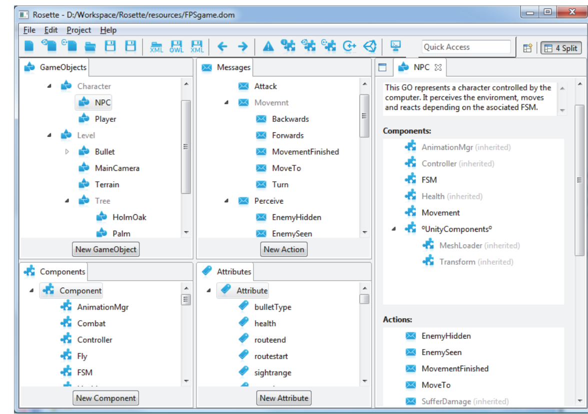

Figure 2: Rosette example

This domain created by the designers represents a definition of the game entities that makes sense from a semantic point of view but, before implementing the game, this domain must be accepted by the programmers. When the programmers accept the given domain they become responsible of adding all the extra information needed for the implementation of the game. They create new entities to represent abstract things such as the camera o triggers, new attributes without semantic interest and possibly new actions and senses that, from the programming point of view, correspond to messages that entities send and accept in the component-based architecture.After completing the population of the domain, programmers distribute the attributes and the functionality among components obtaining components like *Movement* that allows a entity to walk through the environment at a set speed or the *Combat* component that gives the entity the ability to attack in a hand to hand combat.

Figure 2 shows the distribution finally obtained by designers and programmers. Entities shadow in grey (Character, Level or Tree) are only important from the semantic point of view since they will not exist as such in the final game; their main purpose is to enhance the ontology knowledge, but only black entities (NPC, Player, etc.) will be instantiated in the game. The figure also shows some components, actions and attributes in their hierarchies and in the inspector we can see an example of a entity definition with components, actions that is able to carry out, and attributes, which

[2]http://www.w3.org/TR/owl-features/

[3]http://clarkparsia.com/pellet/icv/

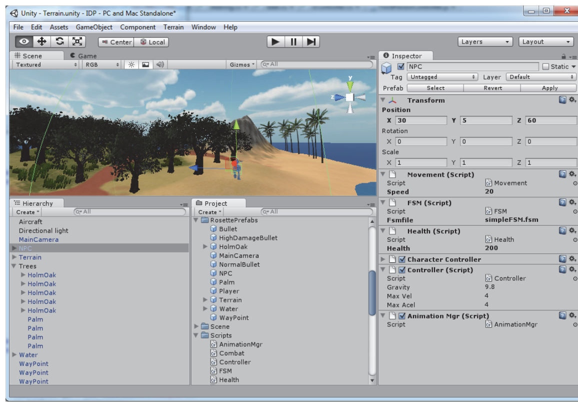

Figure 3: Unity example

are not visible in the figure.

When programmers decide they have finished the technical design, *Rosette* generates the content needed for the creation of the game. Unity scripts are created with the majority of the code and programmers only have to fill some blanks with the concrete functionality. These scripts correspond with the components that are in the ontology. Prefabs, defined as entity prototypes in *Rosette*, are also created in Unity and are used by the designers to populate the level with entities. Figure 3 shows the level created by designers using the *Rosette* prefabs or entities created on the fly by adding components. The figure also shows an NPC in the inspector with the components/scripts that programmers created in *Rosette*.

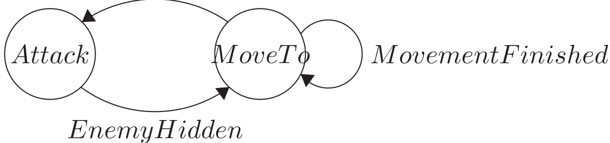

Figure 4: Simple FSM

For this NPC the designers have created a simple FSM (Figure 4) in an external editor to determine how this entity should act during the game execution. The created FSM keeps the NPC moving by the scene and, when the NPC receives an attack or it sees an enemy, it changes its behaviour and attacks the enemy. Then, it returns to its normal behaviour when it loses sight of the enemy. On the other hand, in the level editor (Unity), the designer altered some default values, changing for example the speed value from 5 to 20, to personalize the specific entity and to adapt it to the associated FSM.

So, the level design comprises several task where designers must express their talent putting different building bocks together. As they combine very different pieces of functionality, the level edition usually causes a great number of inconsistencies:

- During the domain design done in *Rosette*, designers and programmers creates attributes that will be added to entities and components. An attribute is defined as a basic o complex type and could have some constraints like an attribute accepting only a set of values or a range of them. In our example, it was set that the *speed* attribute can only adopt values from 0.0 to 10.0.
- Assembling entities on-the-fly, by putting components together, can cause unpredictable failures during the game execution because components sometimes require other components to work properly. For example, the *Movement* component, in order to carry out a *MoveTo* action, need to set the *walk* animation so the entity needs to have another component that is able of setting animations (i.e. *AnimationMgr* component).
- Designers create FSMs and associate them to entities what can cause inconsistencies and errors similar to the previous point. A FSM usually needs to execute actions and a designer can associate a FSM to a entity that is not able to execute some of those actions. The FSM of the Figure 4 need to execute *MoveTo* and *Attack* actions thus at least one component of the entity should be able to carry out those actions.

These inconsistencies constitute terrible headaches for designers because its difficult to find out where the problem is. As problems happen during the execution of the game, the feedback about the problem is non-existent and sometimes the problem is because of technical reasons that designers should not be aware. Because of this, *Rosette* provides designers with the possibility of checking whether their domain is consistent or not with the formal domain they specified before. In the previous section we explained that part of the OWL domain is static (this part related to the attributes, messages and components) and some part is dynamic and created depending of the concrete level (this part related with the FSMs and the entities). Figures 5b, 5c, 5f show examples of the static part of the domain extracted from the game. This code shows how our domain stores the different concepts; interested readers are referred to (Llansó et al. 2011a) to extend this information.

Related to the dynamic part of the OWL domain, Figure 5d is an example of the FSM created by the designer (Figure 4). The FSM is represented as a component that define the events that trigger transitions in the FSM as messages that can be carried out by the fictitious component. In this case the FSM reacts to *EnemySeen*, *EnemyHidden* and *SufferDamage* but, when these events happen, the FSM has to execute an action that should be executed by other component so the entity must contain components that can execute *Attack* and *MoveTo* actions.

As we allow designers to create their own entities with the existing components and they also create FSMs, which in OWL are just new components, *Rosette* must also generate entity definitions dynamically. Figure 5a shows the NPC with its components and attributes while the (Figure 5e) represents a concrete individual where the attribute values are defined as data property assertions. With this semantic information automatically generated when the designer checks the level that he designed, *Rosette* is able to reason and de-

a) NPC entity definition:

```
Character
and (hasComponent some Movement)
and (hasComponent some FSM)
and (speed some float)
and (fsmfile some string)
```

b) Movement component definition:

```
Component
and (interpretMessage some
      MoveTo)
and (speed some float)
```

c) speed attribute range:

```
float
and minExclusive 0f
and maxInclusive 10f
```

d) FSM “component” definition:

```
Component
and (interpretMessage some
      EnemyHidden)
and (interpretMessage some
      EnemySeen)
and (interpretMessage some
      MovementFinished)
and (interpretMessage some
      SufferDamage)
and (isComponentOf only
      (hasComponent some
         (interpretMessage some
          MoveTo)))
and (isComponentOf only
      (hasComponent some
         (interpretMessage some
          Attack)))
and (isComponentOf only
      (fsmfile some string))
```

e) iNPC individual:

```
Object property assertions:
  hasComponent iAnimationMgr
  hasComponent iController
  hasComponent iHealth
  hasComponent iMeshLoader
  hasComponent iMovement
  hasComponent iFSM
  hasComponent iTransform
Data property assertions:
  fsmfile ""
  sightrange 10f
  health "100"^^unsignedInt
  speed 20f
  meshFile "Lerpz"
```

f) Attack message definition:

```
Message
and (target some string)
```

Figure 5: OWL definitions for entities, components, messages and attributes

termine whether the domain violates a property or, on the contrary, the level is consistent with the domain definition. In the example presented during this section designers create a level with some inconsistencies. Figure 5 reveals them:

- The *speed* attribute is set with a not allowed value due to the range was set between 0.0 and 10.0 (Figure 5c) by the game designer but, however, the current value has been set to 20.0 (Figure 5e). In this case *Rosette* informs about the error and the level designer must set a valid value.
- The FSM associated to the NPC will not be able to be executed due to this NPC must have other components with the ability of executing the actions that the FSM has in its states. Although the entity is able to carry out the *MoveTo* action, through the *Movement* component, there is no component in this entity able of carrying out an *Attack* action. In this case *Rosette* just tell the user that he must add a component able of executing an *Attack* and will recommend to use the *Combat* component, the only one of this component collection that is able to do it.

This feedback is very useful to the designer not only when creating a new level but also to check if an old designed level is still consistent when a code and design refactorization is done. This way we promote agile methodologies simplifying the problems between iterations.

Conclusions and future work

The work presents a novel methodology for creating games where programmers and designers collaborate in the process through the use of *Rosette*, a visual authoring tool we have created. Designers specify a declarative model of the game domain, which programmers accept (or suggest changes due to technical reasons), and it serves as the specification that must be implement. The use of an explicit domain (an OWL ontology in this case) helps in several aspects of the development, for example maintaining the semantic cohesion during the development or generating game content in a procedural way.

Tutenel *et al.* (Tutenel et al. 2011) presents a good example of this. They work with an ontology about landscapes, buildings, and furniture that allows to procedurally generating content for a game. The elements they produce are semantically coherent elements of a 3D environment because of they integrate specialized generators (landscape, facade, furniture, etc.) that use the ontology. The work presented here follows a similar philosophy but focusing on the behaviours of the elements and characters of the game instead of creating environments.

After presenting the methodology in a bird-eye view, we concentrate in the benefits that it provides in the level edition. So, in conclusion, we have presented how our methodology and the use of *Rosette* can improve and simplify the work of the game designers. The consistency checking makes designers more comfortable with their job and makes them lose the fear of experimenting with new ideas, as they are less aware of breaking the game or producing bugs because of technical reasons.

As future work we plan to continue developing the methodology, taking more advantage of the semantic knowledge of the ontology and probing how this process works in a bigger game development with more iterations and more complex entities and behaviours

References

Araújo, M., and Roque, L. 2009. Modeling Games with Petri Nets. In Atkins, B.; Kennedy, H.; and Krzywinska, T., eds., *Breaking New Ground: Innovation in Games, Play, Practice and Theory: Proceedings of the 2009 Digital*

Games Research Association Conference. London: Brunel University.

Cook, D. 2007. The chemistry of game design. gamasutra.

Dormans, J. 2012. *Engineering Emergence: Applied Theory for Game Design*. Universiteit van Amsterdam [Host].

Gaildrat, V. 2009. Declarative modelling of virtual environments, overview of issues and applications. Technical report, ToulouseIII University, IRIT, VORTEX Team.

Keith, C. 2010. *Agile Game Development with Scrum*. Addison-Wesley Professional, 1st edition.

Llansó, D.; Gómez-Martín, M. A.; Gómez-Martín, P. P.; and González-Calero, P. A. 2011a. Explicit domain modelling in video games. In *International Conference on the Foundations of Digital Games (FDG)*. Bordeaux, France: ACM.

Llansó, D.; Gómez-Martín, P. P.; Gómez-Martín, M. A.; and González-Calero, P. A. 2011b. Iterative software design of computer games through FCA. In *Procs. of the 8th International Conference on Concept Lattices and their Applications (CLA)*. Nancy, France: INRIA Nancy.

Llansó, D.; Gómez-Martín, P. P.; Gómez-Martín, M. A.; and González-Calero, P. A. 2011c. Knowledge guided development of videogames. In *Papers from the 2011 AIIDE Workshop on Artificial Intelligence in the Game Design Process (IDP)*. Palo Alto, California, USA: AIII Press.

Long, S. 2003. Online product development management: Methods and madness. presentation at the game developers conference, san francisco ca.

Machado, A.; Amaral, F. N.; and Clua, E. W. G. 2009. A trivial study case of the application of ontologies in electronic games.

Neil, K. 2012. Game design tools: Time to evaluate. In *Proceedings of 2012 DiGRA Nordic*. University of Tampere.

Nelson, M. J., and Mateas, M. 2009. A requirements analysis for videogame design support tools. In Whitehead, J., and Young, R. M., eds., *FDG*, 137–144. ACM.

Parsia, B., and Sirin, E. 2004. Pellet: An owl dl reasoner. In *In Proceedings of the International Workshop on Description Logics*, 2003.

Salen, K., and Zimmerman, E. 2003. *Rules of Play: Game Design Fundamentals*. The MIT Press.

Sicart, M. 2008. Defining game mechanics. *Game Studies* 8(2).

Smith, A. M., and Mateas, M. 2011. Answer set programming for procedural content generation: A design space approach. *IEEE Trans. Comput. Intellig. and AI in Games* 3(3):187–200.

Smith, A. M.; Andersen, E.; Mateas, M.; and Popovic, Z. 2012. A case study of expressively constrainable level design automation tools for a puzzle game. In El-Nasr, M. S.; Consalvo, M.; and Feiner, S. K., eds., *FDG*, 156–163. ACM.

Swift, K.; Wolpaw, E.; and Barnett, J. 2008. Thinking with portals. *Game Developer* 7–12.

Tutenel, T.; Smelik, R. M.; Lopes, R.; de Kraker, K. J.; and Bidarra, R. 2011. Generating consistent buildings: A semantic approach for integrating procedural techniques. *IEEE Trans. Comput. Intellig. and AI in Games* 3(3):274–288.

West, M. 2006. Evolve your hierarchy. *Game Developer* 13(3):51–54.

Zagal, J. P., and Bruckman, A. 2008. The game ontology project: supporting learning while contributing authentically to game studies. In *Proceedings of the 8th international conference on International conference for the learning sciences - Volume 2*, ICLS'08, 499–506. International Society of the Learning Sciences.

Zagal, J. P.; Mateas, M.; Fernández-vara, C.; Hochhalter, B.; and Lichti, N. 2005. Towards an ontological language for game analysis. In *in Proceedings of International DiGRA Conference*, 3–14.

Zhang, X.; Tutenel, T.; Mo, R.; Bidarra, R.; and Bronsvoort, W. F. 2012. A method for specifying semantics of large sets of 3D models. In *GRAPP/IVAPP*, 97–106.