RESEARCH-ARTICLE

Explicit domain modelling in video games

DAVID LLANSÓ, Complutense University of Madrid, Madrid, Madrid, Spain

MARCO A. GÓMEZ-MARTÍN, Complutense University of Madrid, Madrid, Madrid, Spain

PEDRO P. GÓMEZ-MARTÍN, Complutense University of Madrid, Madrid, Madrid, Spain

PEDRO A. GONZÁLEZ-CALERO, Complutense University of Madrid, Madrid, Madrid, Spain

Published: 29 June 2011

FDG'11: Foundations of Digital Games
June 29 - July 1, 2011
Bordeaux, France

FDG '11: Proceedings of the 6th International Conference on Foundations of Digital Games (June 2011)
https://doi.org/10.1145/2159365.2159379
ISBN: 9781450308045

Explicit Domain Modelling in Video Games *

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

ABSTRACT

The state-of-the-art in software engineering for game engines, recommends the use of a component-based software architecture for managing the *entities* in a game. A component-based architecture facilitates the definition of new types of entities as collections of *components* that provide basic pieces of functionality, providing a flexible software that can adapt to changes in game design. However, such flexibility comes with a price, both in terms of software understanding and error checking: a game where entity types are just run-time concepts is harder to understand than one with an explicit hierarchy of entity types; and error checking that, in a more traditional inheritance-based architecture, would come from type safety at compile time is now lost. To alleviate these problems, a component-based architecture employs *blueprints*, external data files that specify the particular combination of components for every entity type.

In this paper we propose an extension to the component-based architecture, substituting blueprints with a full fledged domain model in OWL, including a description of the entities, its attributes and components, along with the messages they exchange. We also describe authoring tools for building such a model and show how the model improves software understanding and error checking.

1. INTRODUCTION

In AI terms a domain model is a declarative description, in some formal language, of the entities, classes of entities, and relations among entities for a given portion of the world. A formal domain model for a first person shooter videogame would include the definition of the classes of characters, their attributes, the available weapons and resources, the actions available for characters and their behaviours.

Game development usually starts with an idea that is translated into a design document where game designers describe, using natural language and diagrams, the key aspects of the game, the different characters and elements that appear in the game and so on. During the development process, all this knowledge is transformed into code and assets to create the final game. At the end of the process, the code layer responsible for the management of the game entities contains the *procedural* description of the design document which can be seen as the game's *domain model*.

In recent years, moving from inheritance to component-based software architecture in game engines, has resulted in software whose underlying *domain model* is harder to understand from the source code, since entity types are dynamically defined as collections of *components* that provide basic pieces of functionality. Such definitions come in the form of *blueprints*: data files that specify the particular combination of components for every entity type in a data-driven component-based software architecture.

We propose an extension to the component-based architecture, substituting blueprints with a full fledged domain model in the OWL Web Ontology Language, including a description of the entities, its attributes and components, along with the messages they exchange. Modelling complex domains is not a simple task, and knowledge representation is unlikely to be among the abilities required for a game developer. Nevertheless, we will show how this task can be facilitated through special-purpose authoring tools, and will demonstrate that a formal domain model of the game may have a positive impact on different aspects of the development process, and thus will pay for the effort required in building the model. By representing the game model in OWL, we bring the ability to use reasoning engines that can check domain models for inconsistencies, and thus bring back error checking capabilities that have been lost in game development when moving from inheritance-based to component-based software architectures.

The rest of the paper runs as follows. Next section describes how and why inheritance evolved into component-based architectures, while Section 3 analyses the shortcomings of a component-based architecture. Next, Section 4 presents the general ideas for building a formal model of the game entities in OWL, and Section 5 describes the tool we have developed to facilitate this task. Last section presents related work and concludes the paper.

2. COMPONENT-BASED ARCHITECTURE

Virtually all implementation alternatives for the game logic architecture manage the state of the game through a set of *entities*: self-contained pieces of logic that can perform different tasks such as render themselves, find and follow paths

*Supported by the Spanish Ministry of Science and Education (TIN2009-13692-C03-03)

FDG '11, June 29-July 1, Bordeaux, France

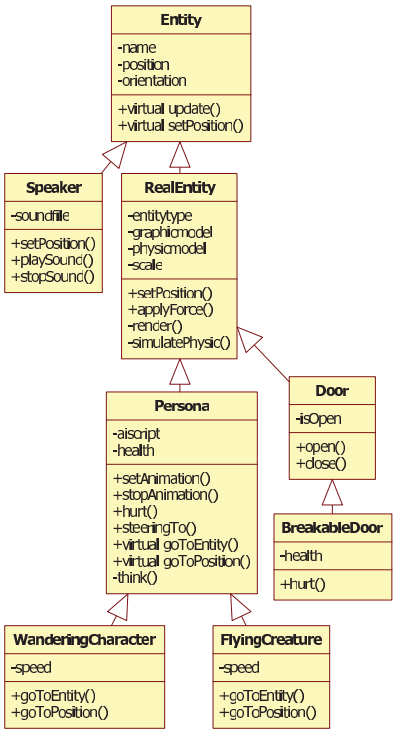

Figure 1: An entity hierarchy

or make decisions [2]. Common entity examples are enemies, players' avatars, interactive items or even props such as doors, trees and fences. Not so obvious entities are *pure logic* elements like camera sequences, waypoint markers, or *triggers* that control plot points in the story. The module implementing and managing the entities in a game is at the core of the game engine and, due to its size and complexity, is responsible for a good piece of a game development effort. To mention just a few examples, a mature game such as Half-Life® dated at 1999 has more than 65,000 non-empty no-comments lines of code on that module, while Far Cry® at 2004 exceed 95,000 lines of C/C++ code[1], even though the majority of the module was actually written in LUA. Two years later, in 2006, came out Gears of War® with 250,000 lines of C++ and scripts [13].

Combining such a software size and complexity with the moving nature of the system specification as determined by an evolving game design, we face the need for a highly flexible and extensible software architecture. When object-oriented languages were finally accepted by game developers, in the early 2000's, they were trading the efficiency of low level C and assembly programming, for the promise of reusability and extensibility that the object-oriented paradigm would bring to cope with increasingly complex software systems. However, the straightforward approach of organizing entities in class hierarchies soon probed too rigid and hard to maintain. The resulting design tends to generate base classes with too many methods and attributes, which

[1]Both lines of code counts were obtained using SLOCCount by David A. Wheeler.

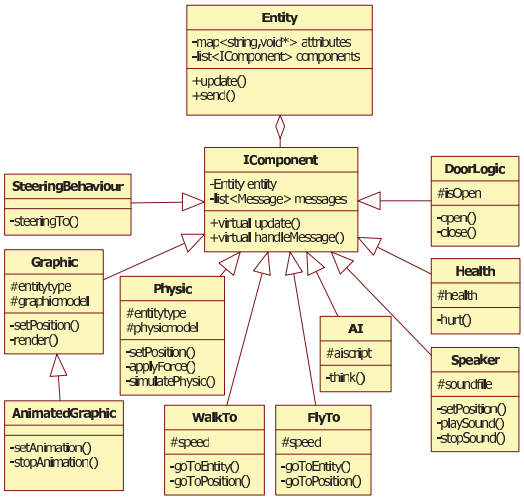

Figure 2: A component-based architecture

are only needed in some of its subclasses. For example, the base class of Half-Life 1 had 87 methods and 20 public attributes while Sims 1 ended up with more than 100 methods.

Still within the object-oriented paradigm but aggressively embracing dynamic object composition instead of static class hierarchies, the component-based software architecture is the design of choice for managing entities in modern video games [11, 16]. Instead of having entities of a concrete class, which define their exact behaviour, now each entity is just a component container where, every functionality, skill or ability that the entity has, is implemented by a component. From the developer point of view, every component inherits from a class or interface (that we will call *IComponent*), while an entity becomes just a list of *IComponent*s.

In order to understand the rationale of a component-based architecture we can see how a class hierarchy such as the one shown in Figure 1 can be transformed into a single *Entity* class with a list of components as shown in Figure 2. Notice how methods from classes in the hierarchy of entities now become methods in components, and, for example, the *think* functionality moves from the *Persona* class to the *AI* component, or *setPosition* moves from the *RealEntity* class to the *Physic*, *Graphic* and *Speaker* components.

In addition to composing functionality from basic building blocks in a way similar to the Decorator design pattern [6], a component-based architecture also incorporates a version of the Command design pattern, which transform method invocation into an object that is passed around [6], and support time-sharing between entities. Notice in Figure 2, that an *IComponent* just has two public methods: *update()* and *handleMessage()*. The *handleMessage()* method is called externally to send it the piece of information; depending on the concrete component, the message will be ignored or stored accordingly. On the other hand, a game engine shares its CPU time among different objects and sub-systems. Periodically, the layer responsible of managing entities takes some of this CPU time and invokes the *update()* method of every active entity, which updates its list of components. When a component is updated, it will process pending messages

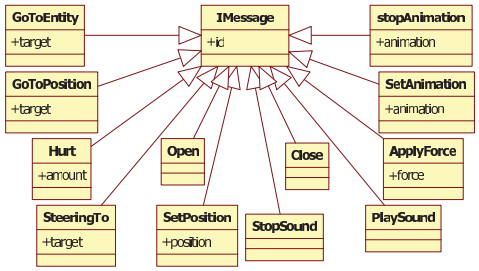

Figure 3: Messages of the component-based architecture

that were requested through *handleMessage()*.

Figure 3 shows the messages needed to execute the functionality assigned to components in Figure 2. For example, when the *AI* component wants to move the entity to a specific place, it sends a *GoToEntity* message to the entity it belongs to. The entity broadcasts the message and the component that implements the ability of movement (i.e. the *WalkTo* component) intercepts the message. When the message gets processed, the *goToEntity* method is executed so this component calculates the path to be followed. While the entity does not reach the destination, the component emits periodically *SteeringTo* messages, to its sibling components, in its *update* method. These *SteeringTo* messages will require the execution of steering behaviour functionality that will guarantee that every point of the path is reached.

Another question that arises when moving from inheritance to component-based systems is where to store the attributes of the entities (such as the *position*). One possible approach is to store the attributes in a list of key-value pairs within the entity (as a map from attribute names to some form of union type, such as the private *attributes* of the class *Entity* in Figure 2). During the initialization, components *register* attributes in the entity to make them public for other components. The access to these attributes is then performed through a method invocation over the entity that, optionally, may inform other components about the modification of the value. The entity itself may also register some general attributes such as *name* or *position*. A second approach is to use the list of key-value pairs in the entity only for those general attributes, while storing additional attributes in the components, and, for example, having the float value that states the walking speed of the entity stored by the *WalkTo* component. Figure 2 shows this mixed approach.

Since entities are now just lists of components, in order to instantiate a particular type of entity we only need to instantiate its constituent components and hand them to an empty entity. Instead of hard-coding instance creation in the code, the actual combinations of components are specified in external files. Such data files, usually known as *blueprints*, are parsed by the game engine at the beginning of its execution, and get used whenever an entity has to be created. Blueprints may also specify default values for the attributes of the entity, as shown in Figure 4, where the blueprint for the entity type equivalent to the *WanderingCharacter* class in Figure 1 is depicted. Blueprints also serve as a way of documenting the type of entities used in the game, moving

```
<blueprints>
  <entity type="WanderingCharacter">
    <component type="Graphic"/>
    <component type="Physic"/>
    <component type="Health"/>
    <component type="AI"/>
    <component type="WalkTo"/>
    <attribute key="entitytype" value="kinematic"/>
    <attribute key="graphicmodel" value="wc.mesh"/>
    <attribute key="physicmodel" value="wc.xml"/>
    <attribute key="scale" value="1.0"/>
    <attribute key="health" value="100"/>
    <attribute key="speed" value="fast"/>
  </entity>
  ...
</blueprints>
```

Figure 4: Blueprint for a wandering character

the domain model implicit in a class hierarchy to external text files, which can be easily modified by non-programmers or generated by special-purpose authoring tools.

3. COMPONENT-BASED ARCHITECTURE ISSUES

It is clear that the component-based architecture promotes reusability and extensibility and therefore it is a good solution from the software engineering point of view. However it is not without some drawbacks. In the next subsections we present three main issues that this approach to fight against the static class hierarchy solution manifests.

3.1 Loss of information about the domain

It is obvious that, despite all the issues class hierarchies manifest, they can be seen as a conceptual entity ontology and therefore they are a non-formal definition of the game model.

Using a component-based system, however, programmers and designers lose this conceptual hierarchy of entities and a big amount of semantic knowledge becomes hidden behind the components' code and the entities definition of the blueprints. The domain model contained in the blueprints is just a plain list of entity definitions with no relationship between them. For example, at a first sight it may be difficult to see the relation between a *BreakableDoor* and another entity *Door* defined as a different set of components because this relationship is not made explicit.

This loss of hierarchy is even more clear when considering abstract classes which are inevitably lost with blueprints.The reason is that the blueprints contains only entities that are meant to be used to create objects. However, when building the class hierarchy there are internal classes that are there just to represent *concepts*. An example is the *Persona* class in the previous Section. As we will never create just "plain" *Persona*s its definition is not included in the blueprints. Therefore, we will have the *WanderingCharacter* and the *FlyingCreature* entities but the relation between them is hard to see because they do not share any common ancestor. The final result of this lack of structure is that the adoption of such design suffered the resistance of some programmers [3, 16].

3.2 Entity inconsistencies at a component level

With a data-driven architecture where is so easy to create new entity types, things may go really wrong, creating inconsistent entities that cannot work at execution time. Before going down into details, we can make an analogy with the use of scripting languages to create behaviours. This languages extract the behaviour out of the game engine which therefore becomes much more independent of the game. However, a buggy script may prevent the game to execute correctly and even stall the application. In that sense an error in a input file provokes a wrong behaviour at execution time. When using the component-based approach, more data is left outside the application and therefore more chances exists to fail the execution because of external aspects. In both cases the core reason is that developers loose the feedback about whether things are correct or not because the game it is always able to start its execution, losing the ideal "if the compiler does not beep, my program should work" [13].

Regarding components, a source of errors come in the form of component dependencies. In a class hierarchy, derived classes depend on ancestor classes to work properly; for example, the *goToEntity* method of the *WanderingCharacter* (Figure 1) use the *steeringTo* of its parent class. In a componentized world all these capabilities have been split in different components. However in some sense there are higher level components that require other components to work properly. Notice that this dependency does not require a particular type of component, but any component that can fulfill a given goal, i.e. can process a given type of message. Following with the example, for executing *GoToEntity* messages (Figure 3), the *WalkTo* component (Figure 2) will need the existence of a component that executes *SteeringTo* messages (i.e the *SteeringBehaviour* component), while, we can imagine a *TeletransportTo* component that can process *GoToEntity* messages without requiring the collaboration of a companion component processing steering behaviour messages: it just moves the entity to its destination.

The key point here is that when a programmer is implementing a component that depends on others, nobody can guarantee to him that every entity that eventually will own this component, is going to also have components that can process its messages.

When working with a traditional inheritance-based architecture, these inconsistencies are checked easier, since the feedback is provided by the compiler. When an entity has not declared a method, which is invoked by another method of the same entity, an error arises at compilation-time. However, in a component-based architecture, when a component sends a message to the entity that it belongs to, and no sibling component can process this message, no error is fired at compilation-time. Even no error is produced at run-time, simply the functionality is never executed. Due to the lack of an explicit error, the task of debugging is harder and it is difficult to know where the reason of this failure is.

3.3 Inconsistencies due to attribute definitions

In a fine-grained approach, entities are populated with attributes as in hierarchies. During the creation of an entity, a list of key-value pairs is given to the entity in order to initialise their attributes. This list results from combining default values defined in a blueprint (Figure 4) and the values for a particular instance of the entity, as specified in a map file. During this initialization of attributes, the entity assumes that these key-value pairs are consistent: values correspond with an expected data-type and is within a valid interval. Although the same problem exists in inheritance-based systems this can be a big source of inconsistencies.

Moreover, what is new in a component-based architecture is the fact that components need that the entity they belong to owns and initialises every attribute that they need. We are talking again about dependencies that are not checked during compilation-time: at run-time components will require attributes that the entity could not have.

3.4 Example

As an example of the all these problems, we can take the seemingly harmless entity definition of Figure 4. Though it is easy to understand the capabilities the entity has, it is hard to see that it presents some of the problems described in this Section:

- The execution of *GoToEntity* messages in the *WalkTo* component is based on the execution of *SteeringTo* messages, which are responsible of moving the entity from one point in the path to the next one at a given speed. Therefore, *WalkTo* depends on a component such as *SteeringBehaviour*. As the entity has the former but does not have the latter, it is inconsistent and it will not work properly.

- The AI component requires the value for an *aiscript* attribute (the file that contains the implementation of the behaviour) that is not provided.

- The *speed* attribute has a string value. However, components that use it may expect a float, as stated in Section 2.

4. USING ONTOLOGIES FOR MODELLING ENTITIES

In a typical game development process, first, the designers reach an agreement about the entities needed for the game. They define every entity and entity feature, which are written in a design document that becomes the first informal domain model. In a component-based system, the programmers agree on how to split those entities in abilities (components) and how these abilities are related (messages). The result is the translation from a natural language model definition into data files organized as blueprints. However, due to the changing nature of videogames, this perfect match between them is quickly broken. As the development goes by, changes are requested by designers that are reflected in messages and blueprints. Such ongoing changes are makes it harder to guarantee that older entities still work and everything is documented up to date.

Our purpose is to provide development teams with tools that allow them to model their game in such a way that reflects the underlying component model. At the same time, we intend to solve the issues of a component model documented through blueprint files as described in Section 3. Our model formalizes the collection of game entities in the OWL web ontology language [2]. Describing the key attributes of game entities using a language like OWL will bring to light a lot of data that is hidden in the code and knowledge that

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

is lost in the translation to blueprints (Section 3.1). Using easy-to-use authoring tools, designers and programmers may define the system that is later stored as OWL definition. This authoring tools may automatically adapt themselves to the changes in the game domain and even act as code generator of component and messages templates in a way that resembles the Model-Driven Architecture [10]. With such a strong domain model, even AI programmers may benefit using the information to increase the capabilities of the AI.

Moreover, having a model of the game entities in OWL allows to use the different tools that are able to reason over OWL representations, checking whether the defined game domain is consistent, regarding those issues described in Subections 3.2 and 3.3. When integrated with the autorhing tools these sanity checks, executed over the OWL domain, let us provide with feedback to the users about what situations should be fixed, thus replacing compile time checking missed in a component-based architecture.

In order to create a formal specification of the entity domain, programmers must define different aspects of it. In this formal domain, these aspects are set in several ontologies, that through the is-a relation, define hierarchies of entities, components, attributes and messages. The root and leave concepts in these ontologies will typically correspond to actual classes in the code. Additionally, inner concepts can be introduced, between the root and the leaves, to structure the model at different abstraction levels:

- **Entity ontology:** Entities must be composed of components and for a fine-grained approach, they have to be populated with attributes. These attributes must be able to be filled in with default values where default values of parent entities could be overwritten in child classes. Maintaining entities in a conceptual ontology helps programmers to rapidly recognise the main purpose of them. Nevertheless this will be only a conceptual ontology and it does not have to be mistaken with a class hierarchy; every game entity will be generic and the difference between entity types is determined by the components they have.

- **Component ontology:** Every component in the ontology must be described with the messages that is able to carry out. Subsection 3.2 said that sometimes there are dependencies between the goal a component has (messages that it carries out) and the subgoals (other messages) that must be carried out by other components to reach the goal. In our OWL domain this turns into restrictions to carry out messages. Furthermore, a component may need that the entity, which it belongs to, has some attributes. This also have to be annotated in the component description. In this case, the ontology will typically match with the C++ representation since a component behaviour can specialise to another one.

- **Message ontology:** A message just represents a requirement of functionality. They are sent to entities and caught by components. As a refined definition, they must be described with attributes that parameterize the required functionality. As in entities, this ontology will be mostly conceptual and only the leaves of the ontology should be implemented.

- **Attribute ontology:** These attributes are the ones used in the previous ontologies. They must be described by a key (just a name) and the type of the values they can take (integer, string, etc.). In addition, these attributes could be described with stronger restrictions such as a set of unique possible values or intervals. However, both these restrictions and the ontology will have nothing to do with the final implementation but they are useful for sanity checking.

Entity, component and message ontologies are represented as *class* hierarchies in OWL, whilst the attribute ontology is represented as an OWL *data property* hierarchy. Since a description of modeling techniques in OWL is out of the scope of this paper, we refer the interested reader to [1].

Entity descriptions use the *hasComponent(?e, ?c)* object property to indicate that *?c* is a component that belongs to the entity *?e*. There also exists the *isComponentOf(?c, ?e)* inverse property. Using the *hasComponent* property and the attributes (described below), entity types are described like in Figure 5a, where a *WanderingCharacter* is a *Persona* that has a *WalkTo* component and a *speed* attribute.

For defining a component, the *interpretMessage(?c, ?m)* object property is used to indicate that the component *?c* can carry out the message *?m*. We also have the *isMessageInterpretedBy(?m, ?c)* inverse property. Components can carry out messages but sometimes they delegate some subtasks. To this end, a component needs other components, in the entity it belongs to, that carry out these subtask messages. Moreover, at run-time, a component may need some attributes from the entity it belongs to. In Figure 5b, the *WalkTo* component is described as a *Component* that interprets *GoToPosition* and *GoToEntity* messages. For these kinds of messages, the *WalkTo* component just calculate the path of the movement whilst the movement itself, through the different points of the path, is delegated. So *GoTo* messages are only carried out if there is another component that is able to interpret *SteeringTo* messages. Furthermore, *WalkTo* can only work with to entities that have a *speed* float attribute.

Messages just require functionalities and they can be populated with attributes in a fine-grained approach. The *GoToEntity* message, defined in Figure 5c, is a message responsible for moving the entity to a *target* destination with a predefined *speed*.

The last kind of elements that can be described are the attributes. As data types, intervals of basic types such as float, string or integer, can be defined but also the range can be reduced to a list of possible values. In Figure 5d, the range of the *entitytype* attribute can only take one of the *static*, *dynamic* or *kinematic* values.

At this point, entities can be modelled in OWL, but no default values are set to them. For this end OWL individuals are used. Every entity, component and message has an equivalent individual but with the name starting with "i" (i.e. Door and iDoor). Individuals let us add object and data properties to them. In Figure 5e, we have the *iWanderingCharacter* individual, which is an instance of the *WanderingCharacter* entity type, and corresponds to the blueprint in Figure 4.

a) WanderingCharacter definition:

```
Persona
and (hasComponent some WalkTo)
and (speed some float)
```

b) WalkTo component definition:

```
Component
and (interpretMessage some
      GoToPosition)
and (interpretMessage some
      GoToEntity)
and (isComponentOf only
      (hasComponent some
        (interpretMessage some
          SteeringTo)))
and (isComponentOf only
      (speed some float))
```

e) iWanderingCharacter individual:

```
Object property assertions:
  hasComponent iGraphic
  hasComponent iPhysic
  hasComponent iHealth
  hasComponent iAI
  hasComponent iWalkTo
Data property assertions:
  graphicmodel "wc.mesh"
  physicmodel "wc.xml"
  entitytype "kinematic"
  scale 1f
  health "100"^^unsignedInt
  speed "fast"
```

c) GoToEntity message definition:

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

d) entitytype attribute range:

```
string and {"static","dynamic",
            "kinematic"}
```

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

5. ROSETTE

As shown in the previous Section, the task of creating a formal domain in the OWL language is not that easy. OWL is an expressive general-purpose language for knowledge representation. Nonetheless, this flexibility comes at a price; OWL is difficult to use and let us define the same thing in different ways. Due to these reasons, we want to restrict its use to the guidelines of the previous Section. As a result of this, we propose the use of a specific-purpose tool, *Rosette*, that let programmers design entities, components, messages and attributes in a visual way. This tool will be then the one responsible of supporting the definition of a domain model in OWL, and connect to a reasoning engine that will provide feedback to the end user.

Rosette (Fig 6), our game domain editor, is a drag&drop and easy-to-use tool. This visual tool let programmers create the four ontologies without writing a single line in OWL. The *Rosette* editor is split in three main frames. The first frame, located on the top left of the editor, contains the four ontologies in different tabs: entities, components, messages and attributes. In this frame, a programmer can create new elements, give them a name and set them in the correct place of the ontology. Whenever an element is double clicked, the element description is opened in the second frame of the editor (properties), located on the right side. Editing the properties of the different elements is as simple as dragging other element from its ontology and dropping it in the properties frame. This way the programmer can add components or attributes to entities or attributes to messages. Messages that a component can execute or attributes they need are added in the same way to components, where message dependencies can be added to previously added messages, by dropping the dependency over the message that the component can execute. These actions not only add the dropping element but also provide some feedback to the user. For example, when adding components to entities, actions they execute and needed attributes are automatically displayed in the entity properties sheet; on the other hand, when attributes are added, they are immediately ready to be filled in with default values. Other features can be also defined using the corresponding field in the properties frame. For example, in order to define an attribute, its type is selected from a list box, whilst restrictions of possible values are added intuitively as shown in Figure 7. The last frame is located in the bottom left of the editor and is used for giving feedback to the end user.

With the purpose of giving feedback to the end user, *Rosette* use the *Pellet Integrity Constraint Validator (Pellet ICV)*[3]. This prototype extends the Pellet reasoner [8] 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 in *Rosette* to provide the correct feedback to the end user.

In order to prove the system, let us present some inconsistent situations. For this purpose, we have modelled the *WanderingCharacter* entity, previously defined in Figure 4 with a typical blueprints file. This entity, modelled with *Rosette* and then translated to OWL, just wander over the environment. The *WanderingCharacter* definition in OWL is shown in Figure 5. For better understanding the inconsistencies, other definitions of Figure 5 are also needed as well as the *AI* component definition, which can only belong to entities with an *aiscript* attribute:

```
AI Component superclasses:
  Component
  and (isComponentOf only (aiscript some string))
```

After defining this model, the programmer may want to validate it. In this case, *Rosette* just launches the *Pellet ICV*, with the OWL model file that it has created as a parameter. This model has three inconsistencies, where the obvious one is that the *speed* attribute is defined as float but then it is asserted as the "fast" string. *Rosette* presents this violation to the programmer (bottom left frame of Figure 6) and this way he can fix it.

When this inconsistency is fixed, another one arises. This one leans on the lack of the *aiscript* attribute asserted in the *iWanderingCharacter* individual. *aiscript* refers to the script executed and requested by the *AI* component and the violation, in fact, arises in the *iAI* individual. As in

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

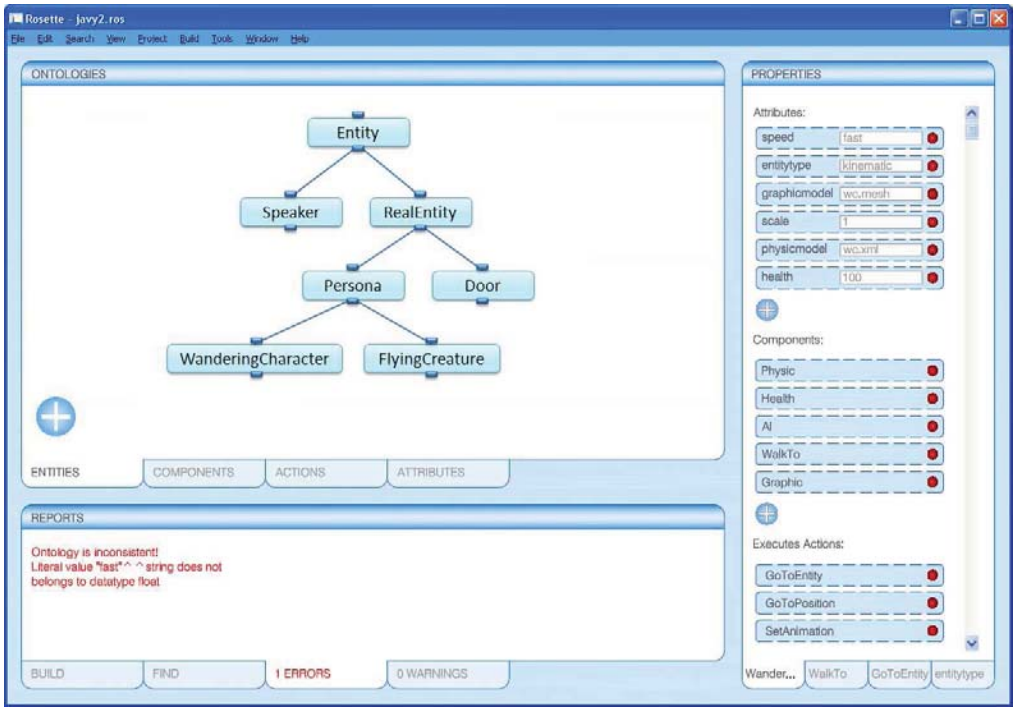

Figure 6: *Rosette*, the game entity editor

the previous inconsistency, *Rosette* provides feedback to the user returning the *Pellet ICV* output:

```
VIOLATION: iAI violates AI subClassOf (Component
and (isComponentOf only (aiscript some string)))
  ...
  NOT INFERRED: iWanderingCharacter type
  (aiscript some string)
    NOT ASSERTED: iWanderingCharacter aiscript _
```

Pellet ICV informs that the *aiscript* attribute has to be asserted in the *iWanderingCharacter* individual. Once this inconsistency is fixed, the third inconsistency is even more difficult to spot. The *WanderingCharacter* entity has a *WalkTo* component that executes *GoToPosition* and *GoToEntity* messages but, as told before, this component delegates the movement by sending *SteeringTo* messages to the entity it belongs to. So, the *WanderingCharacter* entity should have another component that interpreted *SteeringTo* messages, but the *WanderingCharacter* does not have it. This violation is fixed by adding the *SteeringBehaviour* component to the entity.

6. RELATED WORK AND CONCLUSIONS

Current videogames involve a big quantity of code distributed among components. Usually, the bigger the project is, the more complicated entities it involves and, consequently, more components constitute them. In this scenario, some dependencies mentioned in Section 3 are even more relevant, therefore, both industry and academia are working to solve them.

Unity3D [14] is a game engine that let programmers and designers create games using a visual interface and some scripts. Unity3D provides programmers with a simplified way of dependency sanity checking, specifying dependencies between components in the same entity. In a similar approach, Erick B. Passos et al. [9] has proposed a way for checking these dependencies at run-time. Unfortunately the solution requires the use of introspection and attribute annotations available in languages like Java and C# but missing in C++, the mainstream language for building games.

Moreover, previous approaches lose the polymorphism of hierarchical entities. When they impose dependencies between components instead of subgoals that should be carried out, they are not taking into account that actions may be executed in different ways (such as *goToEntity* action). In our previous work [7] we propose a reflective component based system that checks whether an entity is able to execute every action that it is supposed to. It is done by looking for what message dependencies have a component to carry out another message. Nevertheless it comes at a price and involves programmers to write some extra code, every time they create a component.

On the other hand, there are some related work, which are not focused on videogames, in using ontologies in the context of Software Engineering. In some way, *Rosette* is based in a Ontology Driven Architecture methodology [15] that uses OWL to enabling validation and automated consistency checking [18], but also creates a formal domain model to enable the dynamic use by other components and applications in the future. These features used in *Rosette* are also presented in [12] as concrete approaches for using ontologies in the Software Engineering Lifecycle.

Regarding the use of ontologies in games, some efforts has been made to use them for embedding information in the

Figure 7: Attribute properties in *Rosette*

game world. Having semantic knowledge about the entities of the game available, better and more reusable AIs can be build over the top of the game domain more easily [4, 5, 17].

The example presented along the paper is part of a real ontology modelled with *Rosette* for the serious game Javy 2. This OWL domain has 13 different entities, 17 components, 25 messages and 17 kinds of attributes, all of them highly interrelated through object properties. Moreover, the validation of the ontology is carried out in a reasonable time (about 5 seconds).

In conclusion, having a formal domain let us alleviate the problems of a component-based architecture, recovering the conceptual entity ontology and validating inconsistencies. However, it just lays the foundations of future incomes during the game development. OWL domains in videogames bring a lot of data to the light and open the door for future researches. For example, although *Rosette* already provides textual feedback to the end user, this feedback is not as clear as it could be. Now, the feedback is just the output given by *Pellet ICV*. An area to explore is how we could turn it into natural language sentences and visual warnings shown in the ontologies (such as to stress inconsistent nodes).

As future work we are working in how to maintain the relationship between the game code and the OWL knowledge base during the game development, because the relationship is primarily manual today. For that reason we are looking for automatic code generation that, using the OWL definitions, could create the generic entity, every message and the skeleton of every component, filling in some of its methods but also make it flexible to be adapted against changes in the OWL formal domain. Another direction for future research is to extend *Rosette* in order to turn the current entity domain tool into a level editor where the created entities can be positioned within a game level or map.

7. REFERENCES

[1] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[2] S. Bilas. A data-driven game object system. In *Game Developer Conference*, 2002.

[3] M. Chady. Theory and practice of game object component architecture. In *Game Developers Conference*, 2009.

[4] G. M. Y. Frederick W. P. Heckel and D. H. Hale. Influence points for tactical information in navigation meshes. In *international Conference on Foundations of Digital Games(FDG)*, 2009.

[5] G. M. Y. Frederick W. P. Heckel and D. H. Hale. Rapid development of intelligent agents in first/third-person training simulations via behavior-based control. In *Behavior Representation in Modeling and Simulation Conference (BRIMS)*, 2010.

[6] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, Massachusetts, USA, 1995.

[7] D. Llansó, M. A. Gómez-Martín, and P. A. González-Calero. Self-validated behaviour trees through reflective components. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, Stanford, California, USA, 2009. AAAI Press.

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

[9] E. B. Passos, J. W. S. Sousa, E. W. G. Clua, A. Montenegro, and R. L. Murta. Smart composition of game objects using dependency injection. *ACM Computer in Entertainment*, 7(4), 2009.

[10] O. Pastor and J. C. Molina. *Model-Driven Architecture in Practice. A Software Production Environment Based on Conceptual Modeling*. Springer Verlag, 2007.

[11] B. Rene. *Game Programming Gems 5*, chapter Component Based Object Management. Charles River Media, 2005.

[12] S. Seedorf, F. F. Informatik, and U. Mannheim. Applications of ontologies in software engineering. In *In 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006), held at the 5th International Semantic Web Conference (ISWC 2006)*.

[13] T. Sweeney. The next mainstream programming language: a game developer's perspective. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'06)*, page 269, New York, NY, USA, 2006. ACM.

[14] U. Technologies. Unity, 2011. http://unity3d.com/.

[15] P. Tetlow, J. Pan, D. Oberle, E. Wallace, M. Uschold, and E. Kendall. Ontology driven architectures and potential uses of the semantic web in software engineering. W3C, Semantic Web Best Practices and Deployment Working Group, Draft (2006).

[16] M. West. Evolve your hiearchy. *Game Developer*, 13(3):51–54, Mar. 2006.

[17] G. M. Youngblood, F. W. Heckel, D. H. Hale, and P. N. Dixit. *Artificial Intelligence for Computer Games*, chapter Embedding Information into Game Worlds to Improve Interactive Intelligence, pages 31–54. Springer, 2011.

[18] X. Zhu and Z. Jin. Ontology-based inconsistency management of software requirements specifications. In P. Vojtáš, M. Bieliková, B. Charron-Bost, and O. Sýkora, editors, *SOFSEM 2005: Theory and Practice of Computer Science*, volume 3381 of *Lecture Notes in Computer Science*, pages 340–349. Springer Berlin / Heidelberg, 2005. 10.1007/978-3-540-30577-4_37.