Iterative Software Design of Computer Games through FCA *

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
email: {llanso,pedrop,marcoa,pedro}@fdi.ucm.es

Abstract. If iteration is the rule in modern software development practices, this is more the case in game development. While the secret recipe for fun in games remains hidden, game development will remain a highly iterative trial-and-error design process.
In this paper we present a semi-automatic process that, through FCA, can assist in the software design of modern videogames. Through FCA we can identify candidate distributions of responsibilities among components, and let the users edit such distributions. We support iteration by facilitating the application of past edits when going through a new iteration of identifying candidate components to accommodate for new version of the game requirements.

1 Introduction

Nowadays, videogames are incredible complex software systems developed after many months (even years) of work. At the same time, they are, for many people, considered a specific way of art, where designers, drawers, 3D modeller and musicians work side by side to create a rich interactive experience. The artistic aspect of videogames turns them into a big challenge in a software engineering point of view, because their requirements are always changing.

One of the most affected modules of this uncertainty is the one responsible of the management of the *game entities* (players, enemies, items, interactive objects, etc.). Traditionally, this piece of code was implemented using a class hierarchy programmed using object-oriented languages such as C++. Modern videogames, however, use a *component-based software architecture* [9, 19, 4, 15]. It is important to mention that in the context of game development this *component-base architecture* term has a different meaning than the one used in Software Engineering. When talking about *components* here we should think of something similar to mixins or traits [7]: they are just small classes that implement specific and ideally independent capabilities in such a way that a game entity is just a collection of this components (see Section 2 for details). Though this may lead to confusion in the rest of the paper we will keep the nomenclature used in game development and we will use the term "*component*" meaning this small classes.

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

ISBN 978–2–905267–78–8, INRIA Nancy – Grand Est and LORIA, 2011.

Even though the use of components promote flexibility, reusability and extensibility, it does not come without costs (see Section 3 for details). One of the main issues is the lack support in the mainstreams languages (mainly C++) that forces programmers to create a complex source code infrastructure that supports components and the ability of composing entities with them. This also leads to the lack of compiler support in the detection of different problems related to the lost of datatype information that a traditional class hierarchy would have exposed immediately to the programmer.

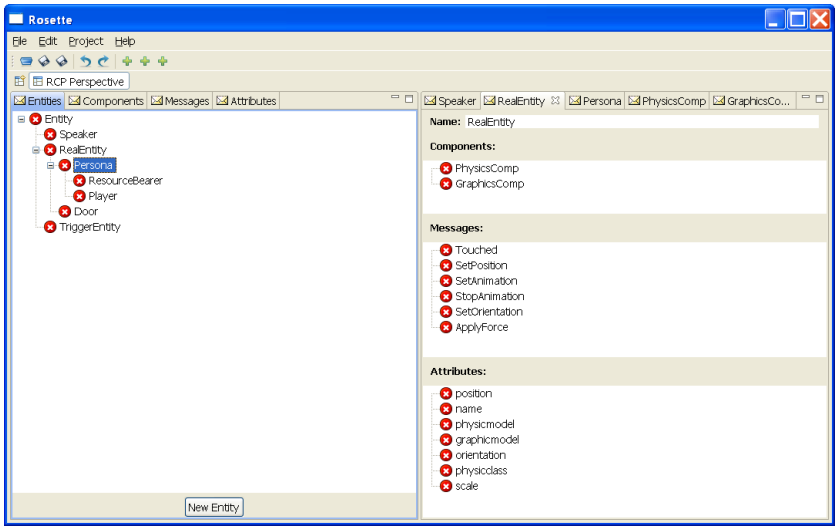

Fig. 1. *Rosette*, the game entity editor

In this paper, we present a new method for easing the design of a component-based architecture. Using a visual tool called *Rosette* [14] (see Figure 1) the user first graphically defines the entity hierarchy (data and actions), a well-known task for game programmers. Using Formal Concept Analysis (FCA) [8], *Rosette* automatically suggests a *component* distribution that will fit the requirements of the provided classical hierarchy (Section 4).

The candidate distribution is shown to the user through a visual interface, where she has the opportunity to modify it (Section 5) before *Rosette* generates a big percentage of the entity manager game code.

Our tool uses OWL (Web Ontology Language[1]) as the underlying representation for the hierarchy and resulting components. It let us track all the applied changes and reason about possible inconsistencies between entities and components that would be hidden to the compiler. This part of the system has been described elsewhere [14].

As said before, videogames specification will surely change in the long run, and the original hierarchy (and component distribution) will need several revisions. As a

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

result of that *agile methodologies*, such as *scrum* [16] or *extreme programming* [1, 2], are taking a big importance in the videogame developments. Agile methodologies give repeated opportunities to assess the direction of a videogame throughout the entire development lifecycle, which is split in iterations or *sprints*. The principal priority is having functioning pieces of software at the end of every sprint to reevaluate the project goals. To this end, *Rosette* allows an *iterative* hierarchy definition during these sprints, remembering all the changes applied to the previously proposed component distributions, and redoing them into the last one (Section 6). This process is possible using the lattices [3] created using FCA and represented with OWL.

2 Component-Based Architecture

Videogames are very complex systems developed by a lot of people who write a big number of code lines during many months. Due to the changing nature of videogames, where requirements are always changing, there are some parts of the game that should be carefully implemented. Specifically, the module responsible of the management of the *game entities* is the most affected module by those continuous changes so it must be flexible enough to be adapted to unexpected changes in the specification and also offer a good way to reuse code for different entities.

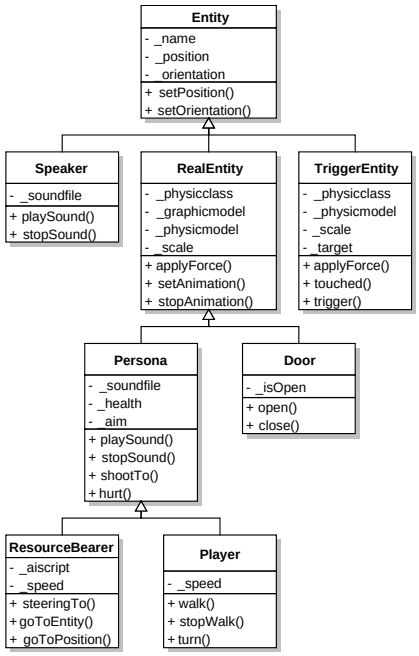

Fig. 2. An entity hierarchy

Traditionally the code layer responsible of the management of the game entities took the form of an inheritance hierarchy of C++ classes. These classes represent the hierarchy of entities and procedures that, in some sense, may be viewed as the actions that these entities were able to perform. Figure 2 shows one of those hierarchies. Each entity class possesses some attributes (data) and methods that code the actions the entity is able to perform. Furthermore, the inheritance propagates features from parents to children; for example, all the attributes and methods of the *Persona* class are also available for *Player*s.

However, the straightforward approach of organizing entities in class hierarchies soon probed too rigid and hard to maintain and evolve. Although well accepted, single inheritance is not expressive enough to handle complex entity distributions. Programmers are usually forced to duplicate code or implement methods too high in the hierarchy [7]. In the last few years, however, the component-based software architecture is the design of choice for managing entities in modern video games [9, 19, 4, 15], embracing this way dynamic object composition instead of static class hierarchies.

Components provide an intermediate level of abstraction between methods and classes by gathering common behaviour and, in this way, they can be seen as mixins or traits [7]. All of them, components, mixins and traits, implement a small set of functionality (usually only one feature) and they invoke other pieces of software that belong to other sibling structures.

The main difference, however, is that in the context of components and games, the entiry hierarchy is flattened and the final result is just an entity class without subclasses that acts as a component container. Every functionality, skill or ability that the entity has, is implemented by a component.

Furthermore, components are more self-contained than mixins or traits since they can have attributes which can specify some states whilst mixins and traits are just a set of methods (we are using the trait concept formalized in [7]). A direct consequence of this is that the entity class can be the same for every entity type, because both their attributes and their functionality are managed by the components. It needs to be just a component container that provides component communication. So, instead of having entities of concrete classes that provide *glue methods* to connect components together like in mixing or traits, the component communication is done like in the *Command* design pattern, where method invocation is transformed into an object that is passed around the components. The piece of information used to execute functionality is called *message* and components decides which messages they will accept to execute the corresponding functionality.

As an example, figure 3 shows the *hand-made* component-based version of the hierarchy of Figure 2. The *Player* entity in the legacy class hierarchy shown in Figure 2 becomes a generic entity containing an instance of the *PlayerControllerComp*, *FightComp*, *PhysicsComp*, *SpeakerComp* and *GraphicsComp* components. Nonetheless, this relationship between entities and components is now done outside the code, usually in plain text files.

Such architecture promotes flexibility, reusability and extensibility but makes the code more difficult to understand, since now the behaviour of a given entity is built at run-time by linking components. The use of a distributed component-based architecture

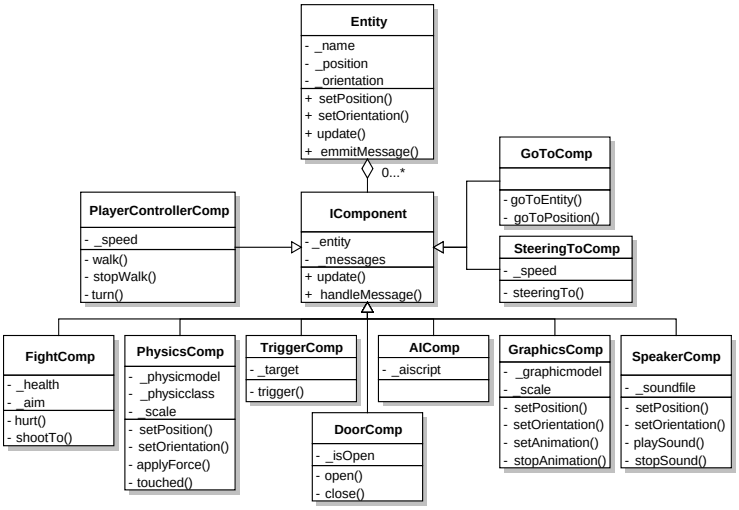

Fig. 3. A component-based architecture

may be confusing for programmers due to the loss of the class hierarchy where the entity distribution is seen at a glance. At the same time, compilers will lose important datatype information, decreasing the amount of compile time errors that will be able to detect.

3 Fighting Against Component Issues

When moving from inheritance hierarchies to component-based systems, the inheritance links are turned into aggregation links where the polymorphism is substituted by a delegation mechanism (delegating in components). This has been identified in [6] as *role aggregation* and it has its own drawbacks that we summarize in:

1. **Loss of information about the domain:** 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. In a component-based system a lot of semantic information becomes hidden behind components and scattered between them. Although in a component-based architecture is so easy to create a new entity, by enumerating in a plain text file which components it will have, entities are not related in between as in hierarchies. Even worse is that abstract classes (such as *RealEntity* in Figure 2) are inevitably lost because only entities that are meant to be used to create objects are modelled. The final result of this lack of structure is that the adoption of such design suffered the resistance of some programmers [5, 19].
2. **Entity inconsistencies:** With a data-driven architecture where creating new entity types is so easy, things may go really wrong, when declaring inconsistent entities that cannot work at execution time. When creating entities in hierarchies, it is obvious that some methods that execute some functionality may depends on other methods in the same entity or in ancestor classes (i.e. the *steeringTo* method of

the *ResourceBearer* in Figure 2 depends on the *applyForce* method of the *RealEntity*). If one invoked method does not belong to the entity, these inconsistency is checked easier, since the compiler provides some feedback. However, when working with components this inconsistency is not checked at compilation-time since method invocation is carried out through message passing. If the *ResourceBearer* has a *SteeringToComp* component (which may execute the *steeringTo* functionality) but does not have the *PhysicsComp* component where the *applyForce* resides, no error is fired at compilation-time. Even no error is produced at run-time, simply the functionality is never executed. On the other hand, when attribute values are also data-driven some error may arise at execution-time when the given value does not correspond with the expected type.

In order to alleviate these drawbacks, we have developed *Rosette* [14], a visual authoring tool that helps experts in the task of designing a game domain with a component-based architecture. Instead of decompose from the very begining all the entities in components, the tool promotes the use of conceptual hierarchies that put entities, components, messages and attributes in order. These conceptual hierarchies recover the possibility of seeing the entity distribution and hierarchy at a glance (solving this way the first of the identified issues) but it is also used to create a knowledge-rich representation of the game domain using OWL that brings a lot of semantic knowledge to light. With the purpose of giving feedback to the end user, *Rosette* use the *Pellet Integrity Constraint Validator (Pellet ICV)*[2] to detect constraint violations in the OWL formal domain and this way detect some domain inconsistencies, such as the ones mentioned previously.

The inner details of the process has been described in [14]. In short it ends with the OWL representation of the game domain and the user being confident about its lack of inconsistences. The next step, the one detailed here, is to find out the best set of components that should be implemented to build the entity hierarchy using a component based architecture. This work alleviate the transition from hierarchies to component-based architectures taking advantage of the modelled domain where experts only describes the entities of the game in a conceptual hierarchy. This way we are facilitating the development of component-based architectures for novice users in this area, more used to hierarchies, but also for expert programmers and designers that may use the system to accelerate the designing process. As we will see in the next section, the analisys of the domain and inference of the set of components is done by using FCA.

4 Generating components through Formal Concept Analysis

When using *Rosette*, the first step is to graphically specify the entity hierarchy of the game. Users must provide both the entities and their features (attributes and methods). For example, they will indicate that a *Player* can *walk* and *turn* whilst a *ResourceBearer* must be able to *goToPosition* or store an *_aiScript*.

Once done, we transform the hierarchy into a *formal context* in order to apply FCA. Entity types (*Player*, *ResourceBearer*) become *formal objects* and features (*walk*, *goToPosition*, *_aiScript*) become *formal attributes*. Therefore, our formal context $\langle G, M, I \rangle$

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

	setPosition	setAnimation	touched	_physicclass	_aiscript	...
Entity	■					...
Trigger	■		■	■		...
Persona	■	■		■		...
ResourceBearer	■	■		■	■	...

Table 1. Partial formal context of the game domain

is built in such a way that G contains every entity type and M will have every functionality and attribute. Finally, I is the binary (incidence) relation $I \subseteq G \text{x} M$, expressing which attributes of M describe each object of G or which objects are described using an attribute. This is filled in by going through the entity hierarchy annotating the relations between entity types and their features and considering inheritance (subclasses will include all the formal attributes of their parent classes). Table 1 shows a partial view of the formal context extracted from the hierarchy shown in Figure 2.

The application of FCA over such a formal context is done by using the Galicia project [18], a project usually used by end-users through their visual interface but that can be used by applications like *Rosette* using its API. The application of FCA gives us a set of formal concepts and their relationships, $\beta(G, M, I)$. Every formal concept represents all the entity types that form its extent and will need every functionality and attributes in its intent. Starting from the lattice (Figure 4) and with the goal of extrapolating the formal concepts to a programming abstraction, a naïve approach is to generate a hierarchy of classes with multiple inheritance. Unfortunately, the result is a class hierarchy that makes an extensive use of multiple inheritance, which is often considered as undesirable due to its complexity.

Therefore, though this approach of converting formal concepts into classes has been successfully used by others [12, 10], it is not valid in our context. After all, our final goal is to *remove* inheritance and to identify *common entity features* in order to create an independent class (a component) for each one. Once done, we will create just a single and generic *Entity* class that will keep a list of components, as promotes the component-based architecture. The addition of new features is done by adding new components to the entity, and these components are independent among themselves. The consequence of this independence is that, now, sharing the implementation of the same feature in different entities is not hard-wired in the code, but dynamically chosen in execution, avoiding the common problems in the long run of the rigid class hierarchies.

Using FCA, we reach this goal focusing not on the objects (reduced extents of the formal concepts), but *in the attributes* (reduced intents). The idea is based on the fact that when a formal concept has a non-empty *reduced intent* this means that the concept contributes with some attributes and/or functionality that did not have appeared so far (when traversing the lattice top-bottom). The immediate result is that the reduced extent

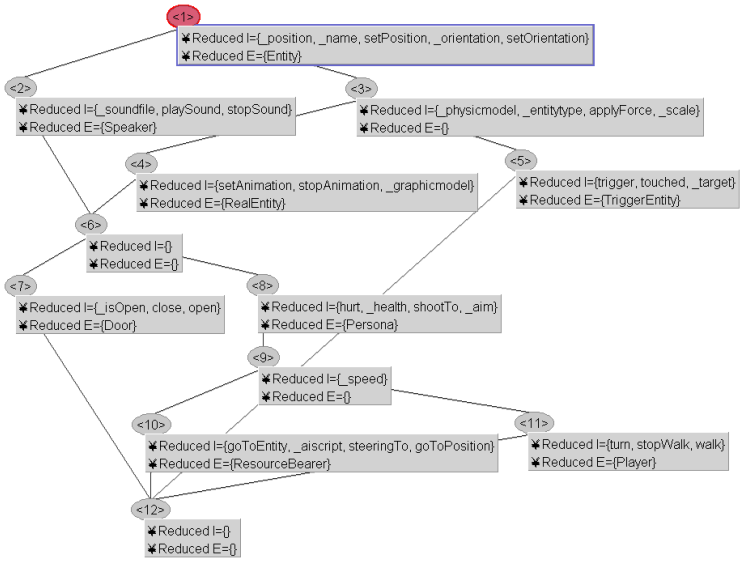

Fig. 4. Concept lattice

of objects differs from the objects in the superconcepts in those properties and it should be consider to build a component with them. At the same time, we will know that all the instances of the entities in the reduced extent of the formal concept will include the new created component.

For example, when analysing the formal concept labelled *11* in Figure 4, our technique will extract a new component containing the features *turn*, *stopWalk* and *walk*, and will annotate that all entities (generic instances) of the concept *Player* will need to include one of those components (and any other components extracted from the formal concepts over it).

The general process performed by *Rosette* with the hierarchy H created by the user is:

G = entity_types(H)
M = attributes_and_messages(H)
I = buildRelation(H)
L = $\beta(G, M, I)$
P = empty component list
for each formal concept C in L
  if C == $\top$ **then continue**
  B_r = reducedIntent(C)
  if B_r is empty **then continue**
  add(P, component(B_r))
end for

All the lines are self-explicative except that with the *add*. The *component* function receives the reduced intent of the formal concept and builds the component representation that has its attributes and functionalities.

In some cases, the top concept ($\top$) has a non-empty intent, so it would also generate a component with all its features (*name*, *position* and *orientation* in our example of Figure 4). That component would be added in *all entities* so, instead of keeping ourselves in a pure component-based architecture with an empty generic *Entity* class, we can move all those top features to it. Figure 5 shows the components extracted from *Rosette* using the lattice from Figure 4. The components have been automatically named concatenating each attribute name of the component or, when no one is available, by concatenating all the message names that the component is able to carry out. For example, let us say that the original name of the *FightComp* component was *C_health_aim*.

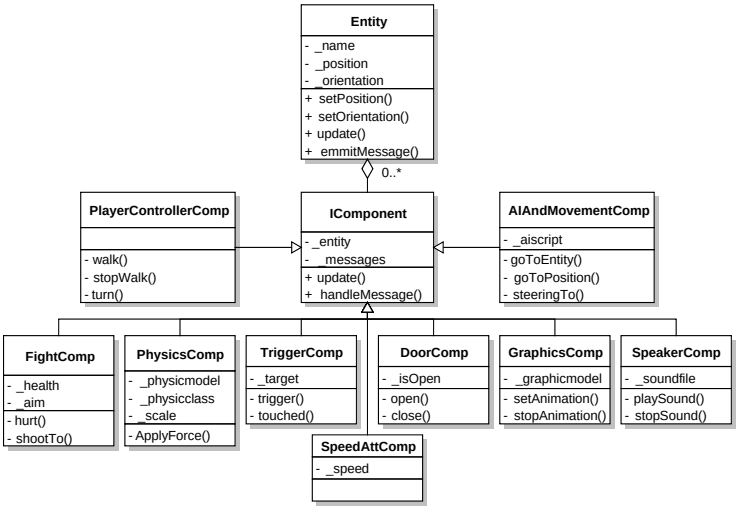

Fig. 5. The candidate components proposed by Rosette

Summarizing all the process, when analysing a concept lattice, every formal concept that provides a new feature (having no empty reduced intent) does not represent a new entity type but a new component. The only exception is the formal concept in the top of the lattice that represents the generic *entity* class, which has data and functionality shared by all the entity types. Both the generic entity and every new component have the ability of carrying out actions in the reduced intent of the formal concept and they are populated with corresponding attributes.

This way, we have easily obtained the candidate generic entity class and components, but we still have to describe the entity types. Starting from every concept which their reduced extents contain an entity type, *Rosette* uses the superconcept relation and goes up until reaching the concept in the top of the lattice. For example, the *Persona* entity type (Figure 4) would have components represented by formal concepts number

8, *4*, *3* and *2* (the number *6* has an empty *reduced intent* so it does not represent a component) whilst the *ResourceBearer* entity type would have the same components but also the number *10* and *9*. Obviously, components of every entity type are stored in the generic entity container represented by the formal concept number *1*.

Keep in mind that the final component distribution does not include information about what components are needed for each entity. This knowledge is not thrown away: *Rosette* stores all the information in the original lattice using OWL, which provides a knowledge-rich representation that will let it provide some extra functionalities described in the next sections.

5 Expert Tuning

The automatic process detailed above ends up with a collection of proposed components with a generated name, and the *Entity* base class that may have some common functionality. This result is presented to developers, who will be able to modify it using their prior experience. Some of the changes will affect to the underlying formal lattice (that is *never* shown to the users) in such a way that the relationship between it and the initial formal context extracted from the class hierarchy will be broken. At this stage of the process this does not represent an issue, because we will not use FCA anymore over it. On the other hand, changes could be so dramatic that the lattice could even become an invalid one. Fortunately, *Rosette* uses OWL as the underlying representation, that can be used to represent richer structures than mere partially ordered sets. In any case, for simplicity, in the rest of the paper we will keep talking about lattices although internally our tool will not be using them directly.

Users will be able to perform the next four operators over the proposed component distribution:

1. **Rename**: proposed components are automatically named according to their attribute names. The first operator users may perform is to rename them in order to clarify its purpose.
2. **Split**: in some cases, two functionalities not related to each other may end up in the same component due to the entity type definitions (FCA will group two functionalities when both of them appears together in every entity type created in the formal hierarchy). In that case, *Rosette* gives developers the chance of splitting them in two different components. The expert will then decide which features remain in the original component and which ones are moved to the new one (which is manually named). Formally speaking, this operator would modify the underlying concept lattice creating two concepts $(A1, B1)$ and $(A2, B2)$ that will have the same subconcepts and superconcepts than the original formal concept (A, B) where $A \equiv A1 \equiv A2$ and $B \equiv B1 \cup B2$. The original concept is removed. Although this is not correct mathematically speaking, since with this operation we do not have concepts anymore, we still use the term in this and in the other operators for simplicity.
3. **Move features**: this is the opposite operator. Sometimes some features lie in different components but the expert considers that they must belong to the same component. In this context, features of one component (some elements of the reduced

intent) can be transferred to a different component. In the lattice, this means that some attributes are moved from a node to another one. When this movement goes up-down (for example from node *9* to node *10*), *Rosette* will detect the possible inconsistency (entities extracted from node *11* would end with missed features) and warns the user to *clone* the feature also in the component generated from node *11*. If the developer moves all the features of a component the result is an useless and empty component that is therefore removed from the system.

4. **Add features**: some times features must be copied from one component to another one when FCA detects relationships that will not be valid in the long run. In our example, the dependency between node *3* and *4* indicates that all entities with a graphic model (*4*, *GraphicsComp*) will have physics (*3*, *PhysicsComp*), something valid in the initial hierarchy but that is likely to change afterwards. With the initial distribution, all graphical entities will have an *_scale* thanks to the physic component, but experts could envision that this should be a native feature of the *GraphicsComp* too. This operator let them to add those "missing" features to any component to avoid dependencies with other ones.

The expert interaction is totally necessary, first of all because she has to name the components but also because the system ignores some semantic knowledge and information based in the developer experience. However, the bigger the example is, with more entity types, the more alike is the proposed and the final set of components, just because the system has more knowledge to distribute responsibilities.

While using operators, coherence is granted because of the knowledge-rich OWL representation that contains semantic information about entities, components, and features (attributes and actions). This knowledge is useful while users tune the component distribution, but also to check errors in the domain and in future steps of the game development (as creating AIs that reason over the domain).

Once users validate the final distribution, *Rosette* generates a big amount of source code for all the components, that programmers will be fill up with the concrete behaviours.

5.1 Example

Figure 5 showed the resultant candidate of components proposed by *Rosette* for the hierarchy of Figure 1, that can now be manipulated by the expert to tune some aspects. The first performed changes are component rename (*rename* operator) that is, in fact, applied in the figure.

A hand-made component distribution of the original hierarchy would have ended with that one shown in Figure 3, that is quite similar to the distribution provided by *Rosette*. When using a richer hierarchy, both distributions are even more similar.

With the purpose of demonstrating how the expert would use the available operators to transform the proposed set of components, we apply some modifications to the automatically proposed distribution in order to turn it into the other one.

First of all, we can consider the *SpeedAttComp* that has the *_speed* attribute but no functionalities. In designing terms this is acceptable, but rarely has sense from the implementation point of view. *Speed* is used separately by *PlayerControllerComp* and

AIAndMovementComp to adjust the movement, so we will apply the *move features* operator moving (and cloning) the *_speed* feature to both components, and removing *SpeedAttComp* completely. This operator is coherent with the lattice (Figure 4): we are moving the intent of the node labelled *9* to both subconcepts (*10* and *11*).

After that, another application of the *move features* operator results in the movement of the *touched* message interpretation from the *TriggerComp* to the *PhysicsComp*. This is done for technical reasons in order to maintain all physic information in the same component.

Then, the *split* operator, which split components, is applied over the *AIAndMovementComp* component twice. Due to the lack of entity types in the example, some features resides in the same component though in the real implementation are divided. In the first application of the *split* operator, the *goToEntity* and the *goToPosition* message interpretations are moved to a new component, which is named *GoToComp*. The second application results in the new *SteeringToComp* component with the *steeringTo* message interpretation and the *_speed* attribute. The original component is renamed as *AIComp* by the *rename* operator and keeps the *_aiscript* attribute.

Finally, although the *Entity* class has received some generic features (from the top concept, $\top$), they are especially important in other components. Instead of just use those features from the entity, programmers would prefer to maintain them also in those other components. For this reason, we have to apply the *add features* operator over the *GraphicsComp*, *PhysicsComp* and *SpeakerComp* components in order to add the *setPosition* and the *setOrientation* functionalities to them.

6 Iterative Software Development with FCA

In the previous section we have presented a semi-automatic technique for moving from class hierarchies to components. The target purpose is helping programmers facing up to this kind of distributed system, which is widely used in computer game developments. Through the use of FCA, this technique splits entity behaviours in candidate components but also provides experts with mechanisms for modifying these component candidates. These mechanisms are the operators defined in Section 5, which execution in the domain alter somehow the underlying formal lattice generated during the FCA process.

Attentive readers will have realized that the previous technique is valid for the first step of the development but not for further development steps. Due to computer game requirements change throughout the game development, the entity distribution is always changing. When the experts face up to this situation, they may decide to change the entity hierarchy in order to use *Rosette* for generating a new set of components. The application of FCA results in a new lattice that probably does not change a lot from the previous one. However, the experts usually would have performed some modifications in the proposed component distribution using our operators. As the process is now repeated, these changes would be lost every time the expert request a new candidate set of components.

Our intention in this section is to extend the previous technique in order to allow an iterative software design. In this new approach, the modifications applied over one

lattice can be extrapolated to other lattices in future iterations. Keep in mind that the domain operators (Section 5) are applied over components that has been created from a formal concept. So, these operators could be applied on similar formal concepts, of another domain, in case that both domains share the part of the lattice affected by the operators.

From a high-level point of view, in order to preserve changes applied over the previous component suggestions, the system compares the new formal lattice, obtained through FCA, with the previous one. The methodology identifies the part of the lattice that does not significantly change between the two FCA applications. This way the tuning operators executed in concepts of this part of the lattice could be reapplied in the new lattice.

The identification of the target part of the lattice is a semi-automatic process, where formal concepts are related in pairs. *Rosette* automatically identifies the *constant* part of the lattice, which for our purpose is the set of pairs of formal concepts that have the same reduced intent. We do not care about the extent in our approach since the component suggestion lays its foundations in the reduced intent. The components extrated from the formal concepts that have not been matched up are presented to the expert. Then she can provide matches between old components and new ones to the considered *constant* part of the lattice.

It is worth mentioning that some of the operators could not be executed in the new domains due to component distribution may vary a lot after various domain iterations but it is just because these operators become obsoleted.

6.1 Example

In Section 5.1 FCA is applied to a hierarchy and the automatic part of the proposed methodology leads us to the set of components in Figure 5. The resultant domain was modified by the expert, by using the tuning operators, and the component-based system developed ends up with the components in Figure 3.

Now, let us recover the example and suppose that the game design has new requirements. The game designers propose the addition of two new entity types: the *BreakableDoor*, which is a door that can be broken using weapons, and a *Teleporter*, which moves entities that enter in them to a far target place. Designers also require the modification of the *ResourceBearer* entity, which must have a *currentEnemy* attribute for the artificial intelligence. The *Rosette* expert captures these domain changes by modifying the entity hierarchy and uses the component suggestion module to distribute responsibilities. The application of FCA to the current domain results in the lattice in Figure 6, where formal concepts are tagged with letters from *a* to *n*.

Comparing the new lattice with the lattice of the previous FCA application (Figure 4), *Rosette* determines that the pairs of formal concepts *<1,a>*, *<2,b>*, *<4,d>*, *<7,f>*, *<9,k>* and *<11,m>* remain from the previous to the current iteration. When *Rosette* finishes this automatic match, the formal concepts that were not put into pairs and with no empty reduced intent are presented in the screen. In this moment, the expert put the formal concepts *<3,c>*, *<5,e>*, *<8,j>* and *<10,l>* into pairs, based on their experience and in the fact that these concepts are very similar (only some attributes

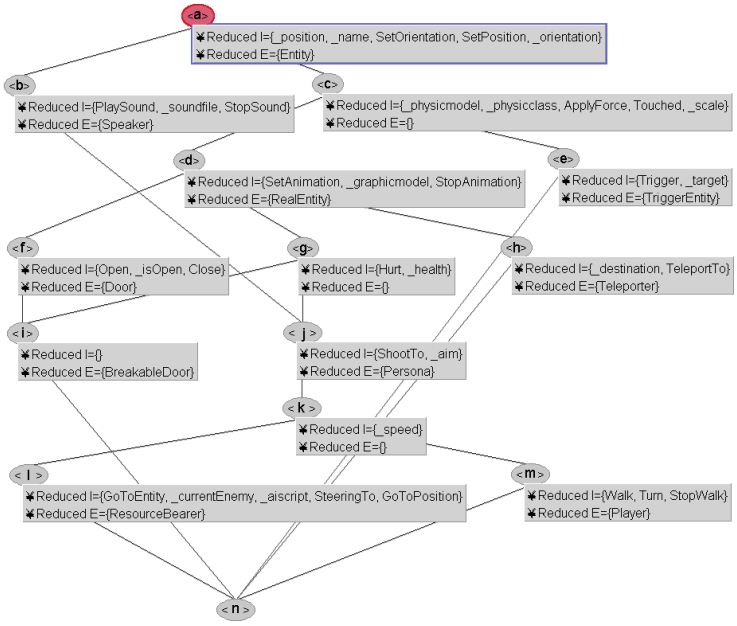

Fig. 6. New concept lattice

changes). Just the *g* and *h* formal concepts have no pairs and will become new components.

So, in these steps, the part of the lattice that does not significantly change has been identified and *Rosette* can extrapolate the modifications applied in the previous lattice to the new one. After applying the operators to the new domain, the new set of candidate components are finally given to the expert. Figure 7 shows these components, where we can compare the result with the components in Figure 5. The general structure is maintained but some actions and attributes has been moved between components. Furthermore two new components have arisen. The stressed features denote new elements (or moved ones) whilst the crossed out features mean that they do not belong to this component anymore (*FightComp*. At this point the expert could continue with the iteration by applying new operators to this set of components (i.e change the auto-generated names of the new components).

7 Related Work and Conclusions

Regarding related work, we can mention other applications of FCA to software engineering. The work described in [12] focuses on the use of FCA during the early phases of software development. They propose a method for finding or deriving class candidates from a given use case description. Also closely related is the work described in [10], where they propose a general framework for applying FCA to obtain a class hierarchy in different points of the software life-cycle: design from scratch using a set

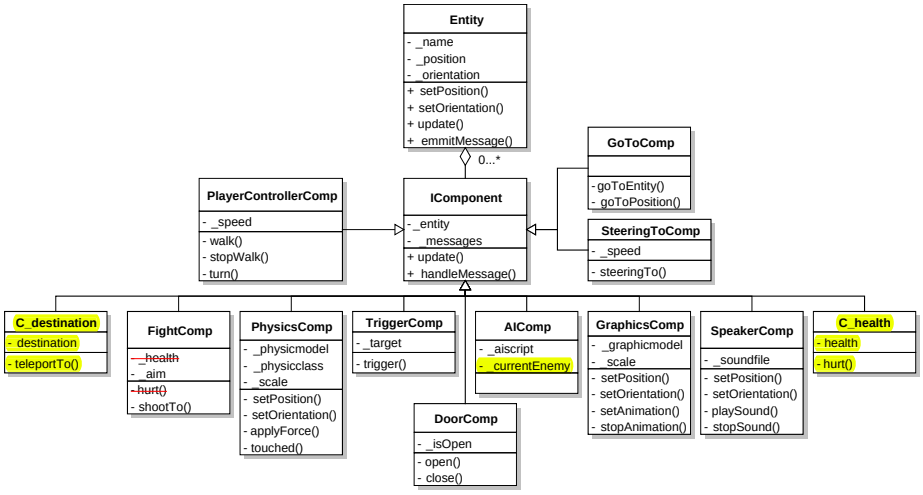

Fig. 7. The new candidate components proposed by Rosette

of class specifications, refactoring from the observation of the actual use of the classes in applications, and hierarchy evolution by incrementally adding new classes. The main difference with the approach presented here is that they try to build a class hierarchy while we intend to distribute functionality among sibling components, which solve the problem with multiple inheritance in FCA lattices.

The process of identifying components with FCA is not very different of identifying traits [13] and aspects [17]. In [13] Lienhard et al. present a process that identifies traits from inheritance hierarchies that is bases in the same principles than our system but is not exactly the same due to components are more autonomous pieces of software than traits. Components save their own state whilst traits are just a set of methods. However, which makes the difference between both proposals is the iterability.

A possible scenario for applying the techniques described in the paper is to re-engineer a game from class hierarchy to components. In the last years, we have been working on Javy 2 [11], a educational game that was initially developed using an entity hierarchy (a portion was shown in Figure 1), and afterwards manually converted to a component-based architecture (Figure 3). When *Rosette* was available, we tested it using the original Javy 2 hierarchy, and the initial component distribution was quite acceptable when compared with the human-made one. We could have saved a significant amount of time if it had been available on time.

In the long term, our goal is to support the up-front development of games with a component-based architecture where entities are connected to a logical hierarchical view. In this paper we have shown how we allow an iterative process when defining the class hierarchy, so operators applied to the early versions of the component distribution are automatically reapplied in the late ones. Nevertheless, more work must be done in the code generation phase to do it reversible. Changes in the autogenerated source code

are still, unfortunately, out of the scope of *Rosette* so they must be manually redone for each class hierarchy iteration.

References

1. K. Beck. Embracing change with extreme programming. *Computer*, 32:70–77, October 1999.
2. K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.
3. G. Birkhoff. *Lattice Theory, third editon*. American Math. Society Coll. Publ. 25, Providence, R.I, 1973.
4. W. Buchanan. *Game Programming Gems 5*, chapter A Generic Component Library. Charles River Media, 2005.
5. M. Chady. Theory and practice of game object component architecture. In *Game Developers Conference*, 2009.
6. M. Dao, M. Huchard, T. Libourel, A. Pons, and J. Villerd. Proposals for Multiple to Single Inheritance Transformation. In *MASPEGHI'04: 3rd Workshop on Managing SPEcialization/Generalization Hierarchies*, pages 21–26, Oslo (Norway), 2004.
7. S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28:331–388, March 2006.
8. B. Ganter and R. Wille. Formal concept analysis. *Mathematical Foundations*, 1997.
9. S. Garcés. *AI Game Programming Wisdom III*, chapter Flexible Object-Composition Architecture. Charles River Media, 2006.
10. R. Godin and P. Valtchev. *Formal Concept Analysis*, chapter Formal Concept Analysis-Based Class Hierarchy Design in Object-Oriented Software Development, pages 304–323. Springer Berlin / Heidelberg, 2005.
11. P. P. Gómez-Martín, M. A. Gómez-Martín, P. A. González-Calero, and P. Palmier-Campos. Using metaphors in game-based education. In K. chuen Hui, Z. Pan, R. C. kit Chung, C. C. Wang, X. Jin, S. Göbel, and E. C.-L. Li, editors, *Technologies for E-Learning and Digital Entertainment. Second International Conference of E-Learning and Games (Edutainment'07)*, volume 4469 of *Lecture Notes in Computer Science*, pages 477–488. Springer Verlag, 2007.
12. W. Hesse and T. A. Tilley. *Formal Concept Analysis used for Software Analysis and Modelling*, volume 3626 of *LNAI*, pages 288–303. Springer, 2005.
13. A. Lienhard, S. Ducasse, and G. Arévalo. Identifying traits with formal concept analysis. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 66–75, New York, NY, USA, 2005. ACM.
14. D. Llansó, M. A. Gómez-Martín, P. P. Gómez-Martín, and P. A. González-Calero. Explicit domain modelling in video games. In *International Conference on the Foundations of Digital Games (FDG)*, Bordeaux, France, June 2011. ACM.
15. B. Rene. *Game Programming Gems 5*, chapter Component Based Object Management. Charles River Media, 2005.
16. K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.
17. T. Tourwe and K. Mens. Mining aspectual views using formal concept analysis. In *Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop*, pages 97–106, Washington, DC, USA, 2004. IEEE Computer Society.
18. P. Valtchev, D. Grosser, C. Roume, and M. R. Hacene. Galicia: An open platform for lattices. In *In Using Conceptual Structures: Contributions to the 11th Intl. Conference on Conceptual Structures (ICCS'03*, pages 241–254. Shaker Verlag, 2003.
19. M. West. Evolve your hiearchy. *Game Developer*, 13(3):51–54, Mar. 2006.