Reflective Components for Designing Behaviour in Video Games

Proyecto Fin de Máster en Sistemas Inteligentes

Máster en Investigación en Informática
Facultad de Informática
Universidad Complutense de Madrid

Autor: David Llansó García
Director: Pedro Antonio González Calero
Co-director: Marco Antonio Gómez Martín

Curso 2008/2009

Documento maquetado con TEXIS v.0.9.

Reflective Components for Designing Behaviour in Video Games

Master Degree's Final Project in Intelligent Systems

Master Degree in Computer Science Research
Computer Science Faculty
Universidad Complutense de Madrid

Author: David Llansó García
Director: Pedro Antonio González Calero
Co-director: Marco Antonio Gómez Martín

Course 2008/2009

Abstract

Developing the AI for non-player characters in a videogame is a collaborative task between programmers and designers and most of the time, there is a tension between them. Nowadays, the industry is making a big effort to separate the collaboration between both groups so programmers need to provide designers with tools which will allow them to design character behaviours. Consequently, the information, about the logic of the game, would be duplicated in the game code and in the field of these tools, leading to errors when someone who modifies the information of one field forgets to replicate it in the other.

On the other hand, *Behaviour Tree*s are an expressive mechanism that let designers create complex behaviours by defining an AI driven by goals, in which complex behaviours can be created combining simpler ones using a hierarchical approach. As well, we propose the use of the *Component-Based Approach*, that is a widely used technique for creating characters in commercial video games avoiding the problems of typical *object-management system*s, which are based on an inheritance hierarchy.

In order to fix the problem of the duplicated and separated information, we propose to gather all the information in components by extending the `IComponent` interface. In this way, entities built by components will be those responsible of providing design tools, or maybe other systems, with all the information that they need for working properly.

Both a general proposal with the basic tenets of our *reflective component-based system* and two concrete systems of this approach are presented. The first one validates associations between *Behaviour Tree*s and characters whilst the second one provides, helped by a planner, different choices to achieve a goal with a character in a concrete scenario.

Key words: Reflective, Components, Authoring, Behaviour, Tree, Videogame, AI, Artificial Intelligent.

Resumen

El desarrollo de IA para NPCs en videojuegos es una tarea colaborativa entre diseñadores y programadores, en la que suele haber conflictos entre los dos grupos. Hoy en día, la industria está haciendo un gran esfuerzo para separar la colaboración entre ambos y para ello, los programadores deben proporcionar herramientas que permitan a los diseñadores crear comportamientos. Por tanto, la información de la lógica de juego estará duplicada en el código y en las herramientas de diseño, conduciendo a errores cuando alguien que modifica la información en un campo olvida replicarla en el otro.

Por otro lado, los *Arboles de Comportamiento* son un mecanismo que permite a los diseñadores crear comportamientos definiendo una IA guiada por objetivos, en la que los comportamientos complejos pueden ser creados combinando otros más simples usando una propuesta jerárquica. También, proponemos usar un sistema basado en *componentes*, lo cual es una técnica muy usada en videojuegos comerciales, evitando así los problemas de los típicos sistemas orientados a objetos con jerarquía de herencia.

Para solucionar el problema de la separación y duplicación de información, proponemos reunir toda la información en los componentes extendiendo la interfaz del `IComponent`. Así, las entidades construidas por componentes serán las responsables de proporcionar a las herramientas de diseño, u otros sistemas, toda la información que necesiten para trabajar adecuadamente.

Se presenta tanto una propuesta general con los pilares básicos del *sistema basado en componentes autodescritos*, como dos sistemas concretos de esta propuesta. El primero valida asociaciones entre *Arboles de Comportamiento* y NPCs mientras el segundo proporciona, ayudado por un planificador, las diferentes maneras que tiene un NPC de alcanzar un objetivo en un escenario concreto.

Palabras clave: Componentes, Autodescriptivos, Autoría, Árbol, Comportamiento, Videojuego, IA, Inteligencia Artificial.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

From the beginning of the game industry and for many years, the most common way of creating behaviour of characters in a game was through programming languages as the rest of the game engine is developed. It has always been a very big problem, and continues to be so, because behaviours are only developed by people with specific knowledge (programmers). When a videogame is being developed, there is a group of people (designers) which are those responsible of design all the concepts of a videogame including the character behaviours. However, designers do not usually have programming skills. Consequently, in order to develop character behaviours, a communication process between designers and programmers must exist. This process can be very long and tedious due to it usually needing many revisions before a behaviour is accepted. Implementing ideas from other people is not easy if they are not perfectly specified.

To deal with the previous problematic situation, the coordination between designers and programmers must be made easier. To this end, programmers should provide designers with some tools that give them the possibility of developing character behaviours for the final game without the need for learning difficult programming languages.

Another existing problem related to the hard-coded behaviours is that they take a really long time to be developed and, as a result of this, videogames with hard-coded behaviours usually have a lack of different character behaviours. Consequently, these games turn into boring games when the user learns how the characters react in every situation. Therefore, if designers were provided with good tools to develop character behaviours, they would develop many behaviours for the same situations and in this way, the process in which the user learned, how the characters react in every situation, would be longer.

Because of this, the game industry has begun to develop new ways of creating character behaviours (Diller et al., 2004). Many games include the option of changing some parameters of the game but this method does not allow designers, or users, the real possibility of designing new behaviours. On the other hand, API's or scripting languages

are much more flexible ways of programming behaviours, but the problem is that designers must learn difficult languages that can lead to errors and they are very difficult to debug, which is not good enough. Other studios use special languages (Hap, ABL, etc.) created with the purpose of modelling behaviours (O.Riedl and Stern, 2006; Ontañón et al., 2007), which are conceptually the same as *Hierarchical Task Networks* that are used in Cavazza et al. (2002a), Cavazza et al. (2002b) and Cavazza et al. (2001). The problem still being the same as in scripting: Designers must learn a new language, and debugging behaviours is a very difficult task.

Due to these reasons, the game industry has introduced tools with graphical interfaces in recent years, in order to simplify the way in which designers develop behaviours and in order to avoid the errors produced during manual programming or scripting. The goal of these tools is to turn the graphical behaviours created by designers into a concrete scripting or special language.

In the majority of games of the last decade, *Finite State Machines* have been the technology of choice for developing character behaviours (Dybsand, 2002; Houlette and Fu, 2004; Rosado, 2004). They are easy to program, fast to execute and game designers feel comfortable designing them so they appear the best choice. Unfortunately, *Finite State Machines* do not scale as well as games need when the NPC's behaviours becomes too complex, and they do not allow easily adding and removing states, or reusing states in other *Finite State Machines* for different behaviours.

1.2 Behaviour Trees

In order to fix these problems, *Behaviour Trees* have been proposed as an evolution of *Finite State Machines* (Isla, 2005). Thus, Behaviour trees are an expressive mechanism that let designers create complex behaviours along the lines of the story they want to tell. They basically define an AI driven by goals, in which complex behaviours can be created combining simpler ones using a hierarchical approach. Nodes in a *Behaviour Tree* represent behaviours, where an inner node is a composite behaviour and a leaf in the tree represents a final action that the NPC must execute.

To promote reusability, behaviours do not include the conditions that lead to transitions. Those conditions are represented as guards in nodes of the tree. In this way, the same behaviour can be used in different contexts with different guards. To further promote reusability, behaviours may be parametrized either hard-coded during design time or during the execution of the tree by other sophisticated behaviours that are executed before. So, in a particular context parameters are bound to actual values in the virtual environment.

There are many kinds of composite nodes described in the literature of *Behaviour Trees*, but in this thesis we only require four kinds of composites: sequences, parallels, static priority lists and dynamic priority lists. A sequence is a list of behaviours that must be executed in the order that they were defined. In the same way, a parallel node is another list of behaviours and all of them have to be executed in parallel at the same time. Meanwhile, static and dynamic priority lists are a composite node that would evaluate

its children guards in order and would activate the first child which its guard was true. The difference between them is that a static priority list only evaluates the guards the first time while a dynamic priority list re-evaluates the guards periodically and if in the execution of a child another more priority child would be able to be executed, the actual branch of the tree would be aborted and the higher priority child would be launched.

For all these reasons we promote the use of *Behaviour Trees*. Nevertheless, they appear as a mechanism too complex for non programmers (Isla, 2005, 2008) so programmers should provide designers with graphical tools for designing them. In the same way, they are difficult to debug so some tests should be checked when a *Behaviour Tree* has been created and in this way, designers would feel more confident with their creations.

1.3 Component-Based Approach

On the other hand, we propose the use of the *Component-Based Approach*, which is a widely used technique for creating characters and game entities in commercial video games. This proposal leaves behind the tendency of typical *object-management system*s, which are based on an inheritance hierarchy, where all different kinds of entities derive from the same base class. In this way, we drop out some of the disadvantages of the class hierarchy that are, among others, an increase in the compilation time (Lakos, 1996), a code base difficult to understand and big base classes (Valve Software, 1998).

In the *Component-Based Approach*, every entity of the game can be seen as a component container in which every component represents a skill or ability that the entity has. These components inherits from the same `IComponent` interface, so for the point of view of the entity, it has a list of `IComponent`s, and every entity is in turn an instance of the same `Entity` class. Due to a component representing an ability, for creating a new entity, designers only need to select which components the entity would have. For example, an entity that represented a door in the game would have a `Graphic` component in order to be rendered, a `Physic` component in order to be collided with other entities and a `Logic-Door` component in order to allow other entities to open and close itself.

Instead of using methods to invoke different functionalities of entities, the *Component-Based Approach* use message passing to execute the different functionalities of the components. Consequently, entities can be seen as message broadcaster, because they are those responsible of resending the messages to every component that they have. For example, if a character wanted to open a door, it would not invoke the `open()` method of the `Door` entity because it does not exist. Instead of that, the character would pass a `Open` message to the `Door` entity and the `Logic-Door` component would receive it and would be the component responsible of opening the door sequentially during successive cycles by sending `Move-To` messages periodically to the rest of the components.

1.4 The Proposal: Reflective Components

Having in mind the increasing tendency of tools for modelling behaviour, and the need of making them better, we propose a new system that eases in the building of new design tools end even it helps to develop systems that provide fixes needed during the execution of the game. The system is called *reflective component-based system* and is based in components that can describe themselves in more than one way.

When tools, with the purpose of creating behaviours, are building, knowing, which kinds of entities the game would have, is necessary. It is obvious because as entities are different in between, they would be provided with different behaviours. So the process through this tools are building is a little bit dangerous because all the knowledge about entities, created by components for the game engine, must be duplicated in the tool. The problem of this is that if a new entity was added to the game or an existing entity was modified, this information should be added also to the tool. It could lead to errors because the person that modify the content of the game can forget to replicate it into the tool and even worse, the person responsible of changing the tool can be another person that has to be informed of the changes.

Summarizing, *Reflective Components* methodology consist of adding new methods to the `IComponent` interface: these methods that allow it to describe itself in the means that the different tools, which use it, need. In this way, the person who added a new entity only should have to make an extra work if he needed to create a new component for the entity and, in this case, he only would need to fill the method, or methods, that the concrete `IComponent` interface required. After that, the concrete *reflective component-based system* would be the one responsible of creating automatically the knowledge that it needed from the components of the entities, or directly use them to extract the knowledge during its execution.

To reinforce our proposal, we have specified two concrete *reflective component-based system*s. The first system (Llansó et al., 2009) is used to validate *Behaviour Tree*s over a concrete entity prior to the execution of the tree. In the process of validating *Behaviour Tree*s, the different branches of the tree are covered. Their leaves (final nodes which contains an action) are checked with the entity, looking for possible failures caused by intrinsic limitations of the entity. Then, depending on the tool that has been proposed, failures would be reported, fixed or ignored. As every component is considered as a skill that an entity has, it is easy to assume that every action of the tree can be carried out by one (or more) specific component. Consequently, the check is done by asking the entity if it is able to carry out the actions of the tree and, in turn, the entity ask it to its components. For this propose, the `IComponent` interface is extended with the `canComponentCarryOut()` method that every specific component class must implement.

With this technique, designers would have an extra check that would assist them, with a tool, when creating behaviour trees and NPCs. Due to the check would be done *before* the execution of the behaviour tree, designers would be more confident about the correct link between NPCs and Behaviour trees. During the execution of the game the system would be used to detect failures in *Behaviour Tree*s as soon as possible in order

to fix them avoiding the game to crash.

The second example (Sánchez-Ruiz et al., 2009a,b) specify a system that would be able to deal with *Behaviour Tree*s by generating different solutions, to achieve goals, having into account the different scenarios in where the entity could stay. This proposal consists on the use of a combination of planning and ontologies. A planner would be able to suggest a set of plans that might be easily turn into *Behaviour Tree*s. Then, the system could be adapted for design tools that helped designers in the task of creating behaviour trees, or the system could be added to the game engine for suggesting alternative behaviours to replace those that were not able of being carried out.

In this system the `IComponent` interface is extended with two methods that are used with the purpose of generating all the information that the planner needs to infer specific plans for the entity that needs a new way to achieve a goal.

The rest of the thesis runs as follow: In Chapter 2 the state of the art of the design tools is raised. Then in Chapter 3, the *Component-Based Approach* is widely explained. After, in Chapter 4 the *Behaviour Tree* concept that is used in the rest of the thesis is described. Next, in Chapter 5 we propose the basic tenets, in which our *Reflective Components* proposal is supported. Following, in Chapter 6, the first concrete *reflective component-based system* is told while the second one is in Chapter 7. Finally, in Chapter 8 the conclusions and the future work is shown. Notice that in Appendix A you will find some code as an example of *Reflective Components* implemented in a serious videogame called Javy 2 that is in develop in our department.

Chapter 2

State of the Art

The traditional method of creating AI, created by programming code, makes the process of creating new behaviours slow and tedious. Furthermore, it usually needs many revisions for removing bugs, and it is limited to people with programming skills. Due to the need of easing the creation of character behaviour, during the recent year, some design tools have arisen. These tools allow people without programming skills creating behaviours. Nevertheless, even there is a lack of good and easy design tools so the industry is in continuous developing looking for new ways to design character behaviours.

2.1 Introduction

From the beginning of the game industry and for many years, the most common way, and maybe the unique, of creating behaviour for game characters was to represent these behaviours through programming languages as the rest of the game engine is developed. It has always been a very big problem, and continues to be so, because in the majority of the cases, behaviours are only developed by people with specific knowledge (programmers). Generally, when a videogame is being developed, there is a group of people that have to design every concept of the videogame, including character behaviours (designers). The problem of this is that designers do not usually have programming skills. So, a communication process between designers and programmers must exist in order to develop character behaviours. This process can be very long and tedious due to it usually needing many revisions before a behaviour is accepted. It is because both translate ideas who one has to a paper is not easy and implementing ideas from other people is not easy if they are not perfectly specified.

In order to avoid the tension produced between programmers and designers, due to programmers envy the freedom that designers have to tell the story that they want and because designers desire the capacity of directly specify behaviours without middlemen, the industry is making a big effort to separate the collaboration between both groups as much as possible. The only possible way to solve this problem is that programmers provide designers with any kind of tool that let them mould the behaviours that they want

to add into the final game without the need, in general, of learning difficult programming languages. These tools are usually called authoring tools.

Nevertheless, the described problem is not the only motivation for creating these tools. An increasingly tendency in commercial videogames, is to provide users with tools that let them create or modify behaviours and characters. It is a fact in which the final users are more interested about the product if it is provided with tools that gives them any type of freedom for expressing their ideas and at the same time, it makes that the contribution offered by users, increases the contents of the game and its continuous evolution. So if programmers were capable of creating tools easy to use by people without programming skills, the game would be easily sold.

On the other hand, users are nowadays more interested in multi-player games played in virtual environments where different games are carried out and where users can face to other users. One of the things that provoke this reaction, is that the hard-coded character behaviours always react in the same way under the same circumstances. So the user finally learns how NPCs act and a fantastic game quickly turn into a bored game. But if people were provided with tools they would develop many behaviours that could be attached to NPCs, so users would never learn how every character acts.

In next Section we are going to revise the authoring tools that the industry has developed (Diller et al., 2004).

2.2 Authoring Tools in Different Fields

During the recent years, due to the increase of the interest in developing mechanism that let people without programming skills creating behaviours different tools that conform or model behaviours have been developed to be used by designers, users and even in the researching field.

Next these tools are presented:

Parametrization

This is the first kind of way to "create" or modify behaviours. Some games use graphic interfaces of easy use that allow people modifying some facets of the character components such as factors that regulate when one or another behaviour is chosen. Nevertheless, the user cannot really develop new behaviours, he can only modify some parameters that change slightly the behaviour; but it is not a new behaviour.

API's

Although this technique is not faithful to everything explained before, because they can be only used by people with programming skills, this technique is a proposal in which authors can create behaviours by using a programming language that is not necessarily the same language that has been used to develop the game itself. In order to create a behaviour, the author would have to implement an interface, with some methods, that

would be connected with the game by means transparent to him. In the researching field this method has been widely used in FPS (First-Person Shooter) games such as Half Life (Valve Software, 1998), Quake 3 or Unreal Tournament, in order to prove some cognitive models that try to imitate human beings.

This mechanism is very useful for researching, but it does not solve the problem given before: programming skills are needed.

Scripts

The use of scripts as a mechanism for authoring behaviours consists in to define the behaviours of the NPCs in a high level language. It can be used after a previous training by people without programming skills. Although it only permits to create a limited behaviours, it is widely use by many commercial games (Quake, Unreal, etc.), which have their own script languages that allow to expand gradually behaviours.

Into the script family, there are more specialized techniques such as *triggers* of condition-response type or *rule-based systems*, a little bit more complicated but, at the same time, better than *triggers*.

The worst problem of this technique is that to debug it, is very difficult because it does not come with a debugger and hard-coded scripts may have many errors and debugging them would be a tedious task.

Plans or Goals Hierarchies

Languages such as Hap or ABL (A Behaviour Language) (O.Riedl and Stern, 2006) are high-level languages designed to create behaviours. In these languages, every activity that has to be executed is considered a goal, and every goal is supported by one or more behaviours that can carry it out. Each behaviour is, in turn, some steps (they can be subgoals) that can be carried out sequentially or in parallel and it has son preconditions that indicate whether the behaviour can be carried out or not.

Different researches use languages such as ABL (O.Riedl and Stern, 2006) or similar ones (Ontañón et al., 2007) in order to create behaviours in a similar way as *Hierarchical Task Networks* (Cavazza et al., 2002a,b, 2001)(HTNs) create it. HTNs also try to achieve goals; in fact, a HTN is a plan with all the possible ways of achieving a goal. HTNs can be explicitly described by means of scripts or they can be generated by tools that automatically generate scripts from graphical interfaces (Kelly et al., 2008) similar to the ones explained later.

Te possibility of committing failures of writing is the worst problem of these techniques because these kinds of failures are terribly difficult to debug.

Graphic Tools

In addition, previous techniques, such as scripting or HTNs, have the inconvenient that developing behaviours with them are a long and not intuitive process because it is not easy to have in mind every relationship between rules, plans and behaviours. One solution

in order to fix this problem is to provide the user with a graphical interface that shows in an easier way these relationships and furthermore it avoids the failures committed by hard-code behaviours.

In examples exposed in Cutumisu et al. (2005, 2006), how to create simple behaviours both reactive and proactive by means of ScriptEase that provides a limited graphical interface can be proved. The process it easier and it avoid failures of handwriting.

Other kinds of more elaborated and easier to use graphical tools are Szilas (2007); Fu et al. (2003). In the first one, behaviour for different entities can be created by means of graphs that represent actions and are linked with others to make sequences. It also allows execution of more than one graph in parallel and makes that graphs trigger other graphs. The system is easy for creating behaviours by non-programmer people but do not create intelligent behaviours.

In the second research, *Behavior Transition Nets* (BTNs) are created in a similar graphic way. BTNs are similar with *Finite State Machine* (FSM) in which actions conditions and evaluation rule are defined. BTNs are a hierarchical mechanism in which an action can be in turn a BTN. It is implemented with a stack to maintain all the states that are in execution due to the hierarchy.

Intelligent Environments

The meaning of these researches is totally opposed to the previous ones because of, instead of model behaviours for NPCs, they create intelligent object that control the NPCs when the objects own them. As an example we can think in *the Sims*, a game in which people can modify NPC behaviours by adding objects.

The problem of this approach is still being the same: users can not create new objects and so that they cannot design new "behaviours".

Learning

There are many games in which their NPCs are trained with various techniques and learning algorithms, but the majority of them freeze this process before they are launched to the market. However, games as *Black & White* use the reinforcement learning, where the user can award or punish its creature and in this way teach it.

Although in the *Black & White* game the result is fabulous, this process could be a little bit bored due to the learning process progress slowly and it must be attended. And furthermore, there exists the possibility that the user tried to teach something and the creature would learn another thing.

Authoring through Behaviour Examples

The motivation of these choices can be summarized in:

1. **Simplicity.** We only need to play a game in order to create a new behaviour.
2. **Humanity.** Behaviours are directly created from a game play.

In these kinds of system (Ontañón et al., 2007; Virmani et al., 2008; Goman et al., 2006; Nakano et al., 2006; Priesterjahn et al., 2006; Priesterjahn, 2008; Priesterjahn et al., 2008), the system has to store a game trace with the information about the actions that the author have been doing during the game. At this moment only is stored the basic actions and then, they will be processed with the purpose of inferring plans that should be similar to the things that the author thought during the game. In this way, some patterns are inferred and in the future the character which would be attached with the inferred behaviour, would act in a similar way that the author did.

It is an interesting way of work for future researches, but in this thesis we are focused into another system closer to the tools and systems that are being used in commercial videogames. Those systems are *Behaviour Tree*s and the *Component-Based Approach* that will be treated in following chapters.

Chapter 3

Components

In the development of a virtual environment, the layer responsible of the management of the entities is usually created using an object-oriented programming language such as C++. Over the years this object-management system has been based on an inheritance hierarchy, where all different kinds of entities derive from the same base class often called `CEntity`.

Nevertheless, as a consequence of this use of inheritance, some problems usually arise. These problems are: an increase in the compilation time, a code base difficult to understand, big base classes, leaf classes with unneeded functionalities and some leaf classes of the hierarchy practically empty.

So, to fix all these problems, we propose the use of the component-based approach, which is also a widely used technique for representing entities in commercial video games.

3.1 Traditional Object-Management Systems

Often, game's entities require lots of functions to define their qualities and their relationships with other entities in the virtual environment, where the NPCs stand out in complexity because they integrate skills such as sensing the environment, perceiving spatial distribution, planning and executing actions and even communicating with other characters.

Traditional object-management systems are often based on inheritance hierarchy where all different kinds of entities derive from the same base class. Classes, directly derived from the base class, are usually also abstract and they represent a split in the tree between classes with different functionalities. During the game's development, some decisions are made about how to split the tree but, due to changeable nature of video games, sometimes those decisions could become bad decisions in the future.

For example, if we had a traditional inheritance tree as in Figure 3.1 and we wanted to add a new different type of enemy called `HumanEnemy`, with the same qualities that the player such as driving vehicles, we would have to botch the tree, in order to allow it, by giving most of the `Player` class content to `Actor` class. This kind of decision would cause that our tree would become increasingly top heavy and it would cause as well that

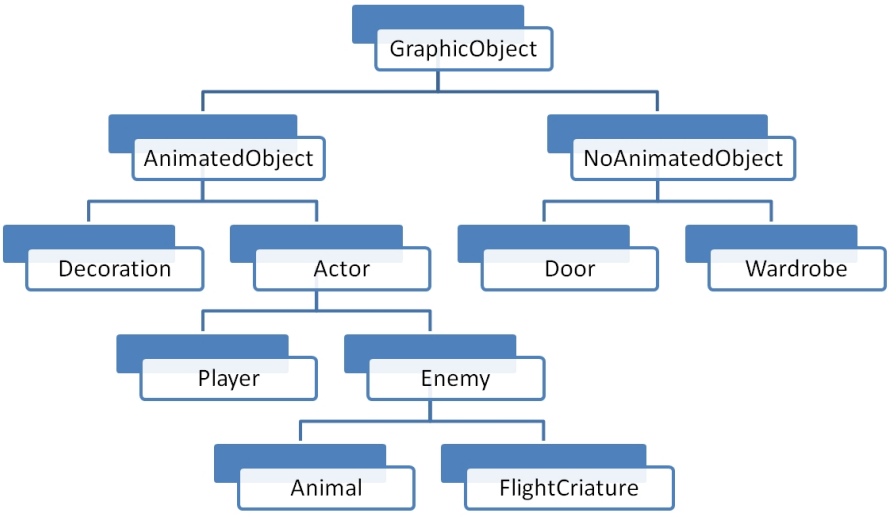

Figure 3.1: Traditional inheritance tree.

classes at the bottom had some unnecessary qualities.

It would be worse if we wanted to allow the `BreakableDoor` class which is able to take damage and to be destroyed like actors. If we wanted to allow it in the tree we would have to botch the tree again because if it inherited from `Actor` class it could not be opened like regular doors but if it inherited from `Door` class it could not be attacked and destroyed.

Some of the consequences of this extensive use of class inheritance are, among others, an increase in the compilation time Lakos (1996), a code base difficult to understand and big base classes. To mention just two examples, the base class of Half-Life 1 Valve Software (1998) had 87 methods and 20 public attributes while Sims 1 ended up with more than 100 methods.

3.2 The Component-Based Approach

Due to all these problems developers tend to use a different approach, the so called component-based systems West (2006); Rene (2005); Buchanan (2005); Garcés (2006), which is a widely used technique for representing entities in commercial video games. With the use of this approach, 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 the `IComponent`, while an entity becomes just a list of `IComponents`.

As entities are now just a list of components and having in mind that every different game entity is just an instance of the same class, which only differs from each others in this list of components that it has, the creation of new kind of entities will be easier.

This is because it does not require any development task and the only the selection of the different skills of the new entity is needed. Consequently it can be done data-oriented using external files such us simple XML files in which there are a list of components per each game entity type. This file is usually called *blueprints* file.

3.2.1 The *blueprints* File

As an example, the previous `HumanEnemy`, which highlighted the disadvantages of the traditional inheritance trees, becomes an entity sharing some components with the `Player` entity: those components that defines the abilities that `Player` had and we wanted to reuse through inheritance. Whereas the majority of the components such as `Move-To`, `Use-Vehicle`, `AnimatedGraphic` would be shared between both entities, components such as `InputDeviceInterpreted` or `BTExecuter` would be own of only one of them. The `InputDeviceInterpreted` component would interpret the keyboard and the mouse input events to make this entity manageable by the user whilst the `BTExecuter` component would permit execute *behaviour trees*, which manage entities as we will see in Chapter 4. In the same way, the `Door` and the `BreakableDoor` would share components but, in this case, the `BreakableDoor` entity would have an extra component (`Life`) with the purpose of giving the entity the ability of being breakable. Figure 3.2 shows a possible description of these entities using a *blueprints* file.

Components may require some external information to configure themselves at the beginning of their execution. For example, the `Graphic` component or the `AnimatedGraphic` component shown in Figure 3.2 would need at least the name of the file with the 3D model and animations. All this information is usually stored in the map file and it is passed to the components during their initialization.

In order to allow fine-grained adjustment of the behaviour (or skills) of different entities, their definition may also require some external information to configure themselves. This information would set the values of different attributes that components would use as parameters of their behaviours. For instance, the `Graphic` component or the `AnimatedGraphic` component shown in Figure 3.2 would need at least the name of the file with the 3D model and animations, or, as an other example, the `Carry` component, which provides the entity with the ability of picking up and transport objects, may use an attribute that specify the strength of the entity.

3.2.2 The *archetypes* File

The initial values of all the parameters of the entities could be specified in the *map* files of the different scenarios. However, if all the parameters had to be specified in every map for every entity, the task could be too tedious. As a consequence of this, the most common parameters of every entity are usually stored in a separated file that can be seen as the file that describes the main characteristics of every entity. This file is usually known as the *archetypes* file, an in this way, in the *map* file, only few parameters should be specified by designers. In Figure 3.3 is shown the *archetypes* file that correspond to the Entities of the *blueprints* file specified in Figure 3.2.

```
<BluePrints>
    ...
    <entity type = "Player">
        <component type = "AnimatedGraphic"/>
        <component type = "Physic"/>
        <component type = "InputDeviceInterpreted"/>
        <component type = "Move-To"/>
        <component type = "Carry"/>
        <component type = "Life"/>
        <component type = "Attack"/>
        <component type = "Use-Vehicle"/>
    </entity>
    <entity type = "HumanEnemy">
        <component type = "AnimatedGraphic"/>
        <component type = "Physic"/>
        <component type = "BTExecuter"/>
        <component type = "Move-To"/>
        <component type = "DummyBehaviours"/>
        <component type = "Carry"/>
        <component type = "Life"/>
        <component type = "Attack"/>
        <component type = "Use-Vehicle"/>
    </entity>
    <entity type = "Door">
        <component type = "Graphic"/>
        <component type = "Physic"/>
        <component type = "DoorLogic"/>
    </entity>
    <entity type = "BreakableDoor">
        <component type = "Graphic"/>
        <component type = "Physic"/>
        <component type = "DoorLogic"/>
        <component type = "Life"/>
    </entity>
    ...
</BluePrints>
```

Figure 3.2: Some entities built by components in a *blueprints* file.

```
<archetypes>
    ...
    <entity type = "Player">
        <attrib name = "life" value = "500"/>
        <attrib name = "strength" value = "strong"/>
        <attrib name = "model" value = "player.n2"/>
        ...
    </entity>
    <entity type = "HumanEnemy">
        <attrib name = "life" value = "100"/>
        <attrib name = "strength" value = "weak"/>
        <attrib name = "model" value = "labourer.n2"/>
        ...
    </entity>
    <entity type = "Door">
        <attrib name = "model" value = "door.n2"/>
        ...
    </entity>
    <entity type = "BreakableDoor">
        <attrib name = "life" value = "50"/>
        <attrib name = "model" value = "door.n2"/>
        ...
    </entity>
    ...
</archetypes>
```

Figure 3.3: An *archetypes* file for the previous entities.

Thus, in Figure 3.3, some parameters of different entities can be seen. A clear example, of how parameters helps in the creation of new entities, is the different values that `Player` and `HumanEnemy` have. As it was told before, `Player` and `HumanEnemy` entities are very similar in between, and they share many components. The difference between them is made through the parameters such as the graphic *model*, their points of *life* or the *strength*.

During the initialization of an entity, it would receive their parameters and, in turn, the entity would pass these parameters to its components. Every different component would take the parameters that it needed, and would keep the entity that launched it. In the case of the `HumanEnemy` and the `Player`, their `AnimatedGraphic` components would take the *model* parameter, so their graphic representation on the environment would differ. in the same way, their `Life` components would take the *life* parameter and consequently the `Player` would have more points of life. Finally, their `Carry` and the `Attack` components would take the *strength* parameter. This parameter would make that the `Player` was stronger and it could take heavier objects and would perform better attacks.

On the other hand, the only difference between both kinds of doors would be that

the `BreakableDoor` entity would have an extra parameter that would set its life into its `Life` component. Their `Graphic` components would have the same model associated so they would have the same graphic representation.

Figure 3.2 lists the section of one of such files that describes the ogre and goblin entity. Entity's description has two main parts, the list of components and the list of default attributes' values. As the figure shows, ogre and goblin entities share some of the components (as `Take` and `TakeCover`). However, they have different attributes that specify different final behaviours. For example, they differ on `strength` attribute that influences the `Take` component. The weapon technology that is used, among others, by `MeleeAttack` is also different (rudimentary versus both rudimentary and elaborate). Finally, the difference of `height` predefines the kind of objects the `TakeCover` component should consider as protections.

In this way, when the game loaded a new level from the *map* file (in which entities would be described as in the *archetypes* file), it would iterate over the list of the level entities and it would use the *blueprints* file for creating entities one by one. After one entity was created, the game engine would have to initialize the entity with its parameters. Thus, to gather all the parameters that the entity needed, the information of the *map* file would be merged with the information of the *archetypes* file, having into account that if the same parameter was in both files, the *map* file would have a higher priority and, in this way, default parameters, which were into the *archetypes* file, could be changed by overwriting them in the *map* file.

3.2.3 Messages

As the components are now generic objects with a common interface independent of their functionality, the usual method invocation is not enough. We cannot have a piece of code calling a method like `MoveTo()`, because no such method even exists. Now, there is a *component* (`Move-To` in Figure 3.2) that is able to move the entity from one point to another, but externally this is just a `IComponent` indistinguishable from other.

The communication is therefore performed in a different way, using message passing. The `IComponent` is viewed as a communication port that is able to receive and process messages. A message is just a piece of data with an identification and some optional parameters (the implementation may vary from a plain `struct` with generic fields used in different ways depending on the type of message to a `CMessage` base class and a hierarchy of messages). Components have a method like `handleMessage()` that is called externally to send the piece of information to it; depending on the concrete component, the message would be ignored or processed accordingly. In this scenario, entities play the role of the broadcaster of messages. Both internal components and external modules may send messages to the entity that are automatically distributed among all its components. An extra advantage of this approach is that it is thread friendly. Let us suppose that a thread executing some code sent a message to a component. Instead of having it immediately processed, it would be stored in the message queue of the component and then when the logic thread launched the update of the entity, all the pending messages

would be treated.

For instance, and following the `HumanEntity` in Figure 3.2, when the `BTExecuter` component (that which provides the entity with the ability to be controlled by a behaviour tree, Chapter 4) wanted to move the entity from one point to another, it would *send* a message to the components of its entity. The component that implemented the ability of movement (`Move-To` in our previous example) would intercept the message, would calculate the path to be followed and would emit periodically `UpdatePosition` messages.

An example of working could be one in which the `HumanEntity` (or the `Player`) entity in Figure 3.2 was attacked. When another entity attacked the `HumanEntity` entity the `Physic` component would be informed by the Physic engine. The `Physic` component would send a `Collision` message with the collided entity as a parameter so the entity would transmit the message through the rest of their components. Consequently, the components that had the ability to carry out that message (in this example it would be the `Life` component) would accept and store it. Then, the `Life` component, during its processing time, would check if the collision was referred to an attack. If this was the case, the component would evaluate if the entity was wounded, and the degree of damage, taking into account some skills of both entities such as attack and defence abilities or strengths. If the attack caused any damage the `Life` component would update its values (such as the life factor) and it would send a `Wounded` (or `Death`) message through the entity. Thus, components such as `Graphic` component would accept and store it for a future process, in which it would play the corresponding wounded (or death) animation.

3.3 A Basic Implementation

With the purpose of gathering all things that have been commented in the previous section, and proposing a formalization of all of them, we have develop an approach of implementation. In Figure 1, three different classes can be seen. In the first one, the `CommunicationPort` class, are all the things related to the message passing. In other words, every thing related to the communication is encapsulated for the `IComponent` interface. This class has two virtual methods that specific components must implement in order to specify with message they will accept (`accept()` method) and how these messages will be processed (`process()` method). These methods are only used by the other two methods of the class in order to store the messages for their future processing (`handleMsg()` method) and to process all the stored messages (`processMsgs()` method).

As it have been commented previously, the `CommunicationPort` class is only the base of the `IComponent` interface. So, this interface add four new virtual methods to control the "life cycle" of the components. It is important to remark that these methods will be probably overwrote by the final component classes. The `activate()` and `deactivate()` methods will be used with the purpose of controlling the activity of the component. For example, these methods are called by the entity when the map, to which the entity belongs, are changed. In this way, entities of maps, which are not in use, can persist without interfering with the execution of the current map.

Moreover, the `spawn()` method will be invoked after the creation of the entity (and

consequently the component) with the purpose of initializing it with the concrete parameters read from the *map* and *archetypes* files. Furthermore, in this method is assigned which entity is the component's owner and, in this way, allow the component for invoking methods of the entity that has this component in its list. Finally, the tick() method will be called periodically in order to check, process and update its values and maybe to send messages to other components or entities.

On the other hand, the `Entity` class is a component container. So, the only propose, of the majority of its methods, is to iterate over its components invoking one of their methods. Examples of this are the `spawn()`, `activate()`, `deactivate()`, `tick()` and `emitMsg()` methods while the `addComponente()` and `removeComponente()` methods are used with the purpose of managing the list of components that the entity has.

```
class CommunicationPort {
    List[Message] _msgqueue;
    ...
    bool handleMsg(Message m) {
        bool acepted = accept(m);
        if(accepted)
            _msgqueue.pushback(m);
        return accepted;
    }
    void processMsgs() {
        while(!_msgqueue.empty()) {
            procesa(_msgqueue.front());
            _msgqueue.popfront();
        }
    }
    virtual bool accept(Message m) {
        return false;
    }
    virtual void process(Message m) { }
    ...
};

class IComponent : CommunicationPort {
    //component owner.
    Entity _ent;
    ...
    virtual bool activate() {
        return true;
    }
    virtual void deactivate() { }
    virtual void spawn(Entity e, List[Parameter] p, Map m) {
```

```
        _ent = e;
    }
    virtual void tick() { }
    ...
};

class Entity {
    List[IComponent] _components;
    Map _map;
    ...
    void spawn(List[Parameter] p, Map m) {
        _map = m;
        for each IComponent c in _components {
            c.spawn(this,p,m);
        }
    }
    bool activate() {
        bool activated = true;
        for each IComponent c in _components {
            activated = activated && c.activate();
        }
        return activated;
    }
    void deactivate() {
        for each IComponent c in _components {
            c.deactivate();
        }
    }
    void tick() {
        for each IComponent c in _components {
            c.tick();
        }
    }
    void addComponente(IComponent c) {
        _components.add(c);
    }
    bool removeComponente(IComponent c) {
        _components.remove(c);
    }
    bool emitMsg(Message m) {
        bool emitted = false;
        for each IComponent c in _components {
            emitted = c.handleMsg(m) || emitted;
```

```
        }
        return emitted;
    }
    ...
};
```

Code Block 1: Pseudo-code of the implementation

Chapter 4

Behaviour Trees

Behaviour trees have been proposed as an evolution of hierarchical finite state machines in order to solve its scalability problems by emphasizing behaviour reuse Isla (2005). Thus, Behaviour trees are an expressive mechanism that let designers create complex behaviours along the lines of the story they want to tell.

Different nodes in a Behaviour trees represent different behaviours, where an inner node is a composite behaviour (corresponding to an abstract state in a hierarchical finite state machines) and a leaf in the tree represents an action (corresponding to a concrete state in a hierarchical finite state machines).

One of the main advantages of having Behaviour Trees is that they explicitly represent the behaviours of the entities. In that sense, behaviour trees can be treated as data and therefore they are able to be analysed in order to extract information both in design time and during its execution.

Nevertheless, at the same time, behaviour trees appear as a too complex mechanism for non programmers Isla (2005, 2008) so programmers must provide designers with design tools and different system with the purpose of reducing the complexity of the mechanism.

4.1 From Finite State Machines to Behaviour Trees

Finite state machines have been the technology of choice for AI in games for decades (Dybsand, 2002; Houlette and Fu, 2004; Rosado, 2004). The reason of this has been that finite state machines are easy to program, fast to execute and game designers feel comfortable designing them. Unfortunately, finite state machines do not scale well when the NPC's AI becomes too complex resulting in a combinatorial explosion of transitions. Besides, Finite state machines do not easily allow either for adding and removing states, or reusing states in different finite state machines. For example, if a new way of attack was developed for being used in different behaviour trees, transitions would need to be explicitly added from all the states in which was valid to go into that state.

In order to overcome the scalability problems in finite state machines, two steps are done to turn finite state machines into behaviour trees:

1. Using procedural mechanisms to determine transitions, turning finite state machines into behaviour lists.

2. Introducing levels of abstraction, turning behaviour lists into behaviour trees.

Behaviour lists represent the AI for the NPC in form of a list of states, in which it can be on. To promote reusability, every state would be provided with a condition (that we will call it *guard*), which have to be checked in order to know whether the NPC can transit to it, and, as well, it would be provided with some algorithm with the purpose of choosing a state when several states that are runnable. So in this way, the transition checking has been removed from the state itself, obtaining states more reusable so that, the same state can be used in different contexts with different guards. As a result of this, in this approach, the different states are called behaviours because they do not need explicit transitions. Consequently, to add a new *attacking behaviour* to a NPC, designers just need to add this state into the NPC's list of states along with a guard that becomes true when that behaviour can be chosen.

The second step to overcome finite state machines limitations is related with the abstraction and hierarchy. This can be solved by turning finite state machines into hierarchical finite state machines (Muñóz Ávila, 2006) or, even better, turning behaviour lists into behaviour trees. The idea of having abstract states that abstract a whole finite state machine, was first proposed by David Harel as part of his *statecharts* specification, a visual formalism. He proposes to extend the state diagrams (the visual formalism for finite state machines), in order to specify complex systems Harel (1987). Hierarchical finite state machines use a stack to store active states, where only the state on the top represents an executable behaviour, and in every cycle it evaluates possible transitions from the active states Richards et al. (2001).

Having in mind the previous paragraph, behaviour lists concept can be extended to final behaviour trees by considering that any behaviour in the list can be itself a composite behaviour with a list of sub-behaviours. The active states of the behaviour trees must be in a branch going from the root to a leaf of the tree (multiple branches if several basic actions can be executed at the same time).

The concept of hierarchy in behaviour trees is crucial to overcome the scalability problem in finite state machines. This is because it introduces a hierarchy of *goals* that allows determining behaviour based on reasoning at different levels of abstraction. Most actions have a primary goal along with a number of additional goals that depend on the action context Atkin et al. (2001).

For instance, the primary goal of the `Move-To` action was to change location from x to y, but in an urban fight scenario the NPC can be moving to get under cover from enemy fire or to assist a fallen comrade. If the behaviour tree was focused only on the primary goal of the actions that are being executed, these actions would sometimes lead to unintelligent behaviour. As an example of this let us suppose that during a movement the NPC was attacked. If the behaviour tree was focused only on the primary goal of the actions, the NPC would continue to move, even when it would be totally destroyed by doing it.

In order to solve this problem, numerous conditional statements could be added to every action specifying all the exceptions to normal behaviour as in finite state machines.

This process would be tedious and could lead to a big amount of errors due to every time that designers wanted to add a new behaviour, they would need to modify many other behaviours. So, instead of adding numerous conditional statements, behaviour trees can handle multiple goals and make them part of a hierarchy, which prioritizes goals higher up in the hierarchy. Returning to the previous example, to stay alive would more important than to move from one point to another. Thus, when the NPC was attacked, some *defence* behaviours would be ready to its execution and due to they would be higher priority, the actual branch would be pruned and one of these *defence* behaviours would be executed.

4.2 Composite Nodes of Behaviour Trees

Although more complex types of selection mechanisms are described in the literature of the behaviour trees, in this thesis we only require four kinds of composite nodes: sequences, parallels, static priority lists and dynamic priority lists.

A sequence consists on some behaviours stored in a list. These behaviours must be defined in a specific order because the purpose of a sequence is to execute the behaviours of the list in the order that they were defined. In similarity with sequences, a parallel node is another list of behaviours with the difference that when it is executed, every child (every behaviour of the list) has to be executed in parallel at the same time.

Meanwhile, static and dynamic priority lists are a composite node that would evaluate its children guards in order and would activate the first child which its guard was true. Each priority list node represents different choices to achieve the same goal, and into these choices, the best choice is that which is the most priority. The difference between them is that a static priority list only evaluates the guards the first time, when the node is invoked to run, while a dynamic priority list re-evaluates the guards of its children periodically and if during the execution of a child another more priority child, which was not runnable when the other child was launched, turn into a runnable node, the actual branch of the tree would be aborted and the more priority child would be launched.

The behaviour selection mechanism has to be completed with an execution model that determines when to re-evaluate guards for candidate behaviours. Typically, guards are re-evaluated periodically after a given number of game ticks, when certain game events occur or when the active behaviour finish its execution. Notice that guards associated to behaviours are not like preconditions that, when they are fulfilled, guarantee its successful completion. These guards indicates, as in abstract state machines Borovikov and Kadukin (2008), that the behaviour can be chosen, although it may terminate with failure or success.

The abstraction given by behaviour trees allow designers to treat a tree as a new complex behaviour that can be later reused in other behaviour trees that cover more cases. In that sense, during the design time, designers would firstly create basic behaviours trees that represent simple behaviours, and then these basic behaviours would be attached to different branches of more general behaviours trees developed for one or more NPCs. In order to have behaviours more reusable, they may be parametrized either hard-coded

during design time, or during the execution of the behaviour tree, by means of other sophisticated behaviours that are executed before. So, in a particular context, parameters may be bound to actual values in the virtual environment.

For example, if we had a `Take` behaviour, the object which was to be taken should be described as a parameter in order to promote reusability. Then, when the behaviour was put in a behaviour tree, the parameter (the object which was to be taken) should be specified either by hard-coding it or by another behaviour that would be executed before with the purpose of filling the parameter. In this way, this behaviour could be used to take whatever we want, and we would not need to develop new behaviours to take different objects.

Notice that behaviour trees represent the behaviour of NPCs, so each behaviour tree in execution is associated to a NPC. Thus, when a behaviour tree is executed by a NPC, an execution context must be created for it. The context of a behaviour tree is made up of a set of variables, in form of *(attribute, value)*. This context is made up of attributes of the *Game State* that can be accessed by the NPC. Different guards and actions of the behaviour tree can sometimes consult or change these variables so every attribute references in the tree must form part of the context.

Generally, a behaviour tree's context contains at least two variables:

- ?*this*: references the NPC executing the behaviour. The attributes of this variable describe its properties (e.g. strength, force, defensiveness, health).

- ?*world*: references the virtual environment in which the game takes place.

In order to allow a behaviour to transmit some parameters to another child behaviour, behaviours can be provided with some input parameters. When a behaviour is invoked, its parameters would be bound to a value, either a literal value hard=coded by designers, or a value that come from the context of the behaviour tree that is still in execution. All of these values will form part of the context of the newly invoked behaviour tree. In the same way, exchange of information can be done in the opposite direction (from children to parents) through the context variables.

In Figure 4.1, for instance, the `Look for a thing` behaviour has two input parameters, `?this` and `?what`, that are bound to the variable `?this` of the `Wood harvesting` behaviour tree context and to the hard-coded value *"log"*. When this behaviour was invoked, it would find the nearest object of the specified kind (in this case a log) and it would store it in the `?this.target` variable of the context. Then, the `Move-To` behaviour is bound with the entity and the selected log in order to move the entity to the log. As well, the `Take` behaviour is bound with these two variables to makes the entity take the log. Finally, the `Move and leave` sequence is bound with some variables in the same way in order to execute two sub-behaviours that makes the entity go to home and leave the log there.

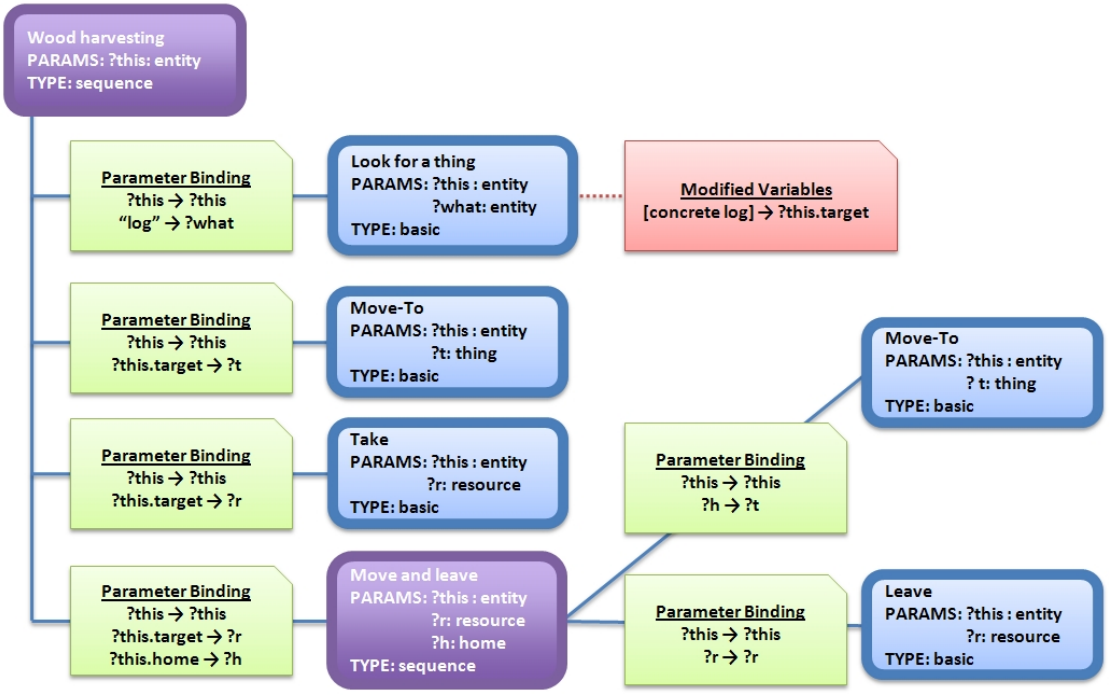

Figure 4.1: Behaviour Tree of "wood harvesting" type.

4.3 Behaviour Tree Executer

Having components (Chapter 3) for creating entities of the virtual environment, when the implementation of the AI of an entity using a behaviour tree has to be tacked, the obvious approach would be to create a new component that would be able to execute a given behaviour tree over that entity. In Figure 4.2 this component is called `BTExecuter`.

This new component can be seen as an interpreter of behaviour trees. It is meant to read the properties of the entity during initialization time and load the behaviour tree file specified on them. It is able also to handle different messages for changing the current behaviour tree in execution; this is useful when designers want to hard-coded a change on the behaviour for the sake of the gameplay.

The composite nodes that compound the behaviour tree are executed accordingly by the component. It tracks the branch of the behaviour tree being executed and checks the condition of the open nodes periodically. When the flow of execution reaches a leaf where a basic action resides, it has to perform it over the entity it belongs to.

These basic actions (such as a `Move-To` action) can be carried out in two different ways. The first approach consists on running it autonomously, sending messages in every tick over the other components of the entity. For example, in a `Move-To` action the `BTExecuter` component may send update messages of positions in every tick.

In the second approach the `BTExecuter` component does not take the responsibility

```
<BluePrints>
    ...
    <entity type = "HumanEnemy">
        <component type = "AnimatedGraphic"/>
        <component type = "Physic"/>
        <component type = "BTExecuter"/>
        <component type = "Move-To"/>
        <component type = "DummyBehaviours"/>
        <component type = "Carry"/>
        <component type = "Skills"/>
        <component type = "Use-Vehicle"/>
    </entity>
    ...
</BluePrints>
```

Figure 4.2: `CHumanEnemy` entity built by components.

of executing any of the basic actions but delegates its execution to other components that the entity must own. In that sense, the atomic actions that appear in the leaves of the tree will be carried out by other components (such as `Move-To` component or `FollowPath` component). Our `BTExecuter` becomes a director of the execution that sends messages with the action information and waits the confirmation messages indicating that executions of these actions has ended, in order of having the behaviour tree execution moves forward.

Our proposal for future chapters is based on this second way. As we have see Section 3 every component represents an ability/skill that the entity has. So, if actions are carried out by different components, abilities and skills of the entity are explicit because we can link behaviour tree actions with components that can carry out those actions. In following chapters it will give us the possibility to applied inference mechanisms that will help us to create systems that detect and fix failures of behaviour trees associated to specific entities.

Let us propose an example in which the `BTExecuter` component has to move the entity from one point to another(Figure 4.2). This component would send a `Move-To` message, with some parameter like the target position, to the entity. The entity would transmit the message through their components and the `Move-To` component would accept and store it. Then the `Move-To` component during its processing time would find the correct path for the movement and it would send `ChangePosition` messages periodically, with parameters such as the new position or the movement type, through the entity and components such as `Physic` component or `Graphic` component would accept and store it for future process in which they would change positions of the physic and the graphic entities and would play the corresponding animation. Finally, when path was completed, the `Move-To` component would send a confirmation message that the `BTExecuter` component would receive to continue the behaviour tree execution.

Chapter 5

Reflective Components

In this chapter the basic tenets, in which our *Reflective Components* proposal is supported, will be exposed. Here it is only explained the base that every *reflective components system* should have. Then, in chapters 6 and 7 two different approaches, which are built on the top of these tenets, will be explained.

5.1 Gathering Information

As we explained before, developing the AI for non-player characters in a video game is a collaborative task between programmers and designers. Most of the time, there is a tension between the freedom that the designers require to include their narrative in the game, and the effort required by programmers to debug faulty AI specified by good story tellers who are not programmers.

Nowadays, the industry is making a big effort to separate the collaboration between both groups as much as possible, so programmers need to provide designers with tools which will allow them to design history plots, character behaviours, etc. Consequently, some of the information, about the logic of the game, which forms part of C++ classes, have to be duplicated in the design field to develop these design tools.

This duplicated and separated knowledge can lead to errors due to the fact that someone implementing a new functionality in C++ classes may forget to replicate this knowledge in the design field of every tool. Even usually programmers who implement game functionalities and programmers who implement design tools are different people. Thus, if there were not a perfect coordination between both parties, some of the implemented game functionalities would be never used by the designers because these functionalities could not be replicated in the design tools and the designers would ignore their existence.

As an example, we can think of a tool to build Behaviour Trees for component entities. In that tool, some extra information about entity abilities or skills would be needed to know which types of actions could be carried out by each entity. So, every time we wish to add a new component/ability to an entity in the implementation field, or every

time we want to add a new entity in the implementation field, we have to redefine the knowledge of these entities for the design field too, to enable designers to use them in the correct way. Therefore, as we explained before, if there was not this perfect coordination between game developers and tool developers, some of the implemented game entities, or entity abilities of those entities (and consequently actions that they are able to carry out), would be never used by the designers because these qualities could not be implemented in this design tool.

For avoiding this tedious, difficult and probably uncoordinated process, in which the information has to be defined in different places, we propose to gather all the information through the *reflexive components*, which are responsible for describing themselves. In this way, all the information about the game entities is stored in the same place: the components. Thus, the one who implemented a new component would be the person responsible for filling some extra methods that would allow one to describe this new ability in the design field.

5.2 A Reflective Component-Based System Develop Methodology

In previous chapters we have defined how entities are built from simple XML files in which they are described as a collection of components. Therefore, our fine-grained approach should consider different limitations due to different parameters that are related to entities and their components. We said that our proposal was to bring all the knowledge together into components, consequently the C++ implemented game entities, with their components, had been specifically described for some tools. So during the execution of these tools, the implemented game entities have to be used. Because of this, these entities have to be initialized during the design time in order to be used by these tools. This is because components are responsible for knowing their abilities, their skills or which actions they are able to do.

Nevertheless these initializations do not have to be full initializations because there would be many unnecessary functionalities programmed for the final game, therefore partial initializations have to be allowed to use the components by the design tools. For example, during the game execution, a `Graphic` component will create a graphic entity into the graphic engine from the model specified to the component. Then the engine will render the model to present it on the screen. However it is not necessary during design time and the component might only need the animation names in order to know which types of animations are able to be played and consequently which types of actions could be represented by the model.

The *reflective component-based system*, which we are proposing, could be used in many ways to support different design tools, and these descriptions would be very close together. Although later we will explain two approaches for two different tools, now we want to abstract all those possible descriptions to propose a *reflective component-based system developed methodology*. Owing to the fact that we explained in a previous section

how to build a *component-based system*, in continuation, differences between a usual *component-based system* and a *reflective component-based system* will be stressed to see how our system extends the *component-based system*.

The First difference between a usual *component-based system develop* and a *reflective component-based system develop* is their initialization. In our proposal, there is a difference between design and execution initializations as we explained before, thus when a *reflective component* is developed partial or full initializations have to be allowed. It could be done by adding a new method, such as `spawnInDesign()`, to the `IComponent` interface but, depending on the functionality that components share between the design time and the game execution, it could be done in the same `spawn()` method avoiding duplicated code between both methods.

Some tools may need to extract component information once to collect all the knowledge and to process it. So, components are only needed during the initialization of the tool. However, other tools may need entities and their components during their whole life due to the fact that they would need to check the entities' abilities during the tool's working time. Consequently, one or more methods have to be added to the `IComponent` interface, to allow the *reflective components* to describe its abilities, skills or which actions they are able to do, in the way that the tool needs.

Tools may want to directly ask components, but usually they want to ask entities about their abilities, skills or which actions they are able to do, because these tools usually ignore that an entity is a collection of components. So we have to provide entities with one or more methods that iterate over the list of components collecting the information provided by them. In Code Block 2 a general view of methods that have to be implemented for developing a *reflective component-based system* can be seen. Then each concrete component, which would have to inherit from `IComponent` interface, has to be implemented with their specific `spawn` and `getinformation()` methods. Let us recall that different `getinformation()` methods can be added to give the information in different ways, both for the same tool and for more than one tool.

```
class IComponent {
    ...
    virtual void spawn(bool full);
    virtual Info getInformation(Data consult)=0;
    ...
};

class Entity {
    List[IComponent] _components;
    ...
    Info getInformation(Data consult) {
        Info info;
        for each IComponent c in _components {
            info = merge(info, c.getInformation(consult));
```

```
        }
        return info;
    }
    ...
};
```

Code Block 2: Pseudo-code of the implementation

In following chapters two different proposals of *reflective component-based system*s will be shown for two different processes. Firstly, a *reflective component-based system*, which helps us to validate Behaviour Trees over different entities, will be seen. The system validate Behaviour Trees by checking which actions of the tree can be carried out by the specified entity. Secondly, another *reflective component-based system*, which helps designers in the process of building Behaviour Trees for concrete scenarios, will be seen. In this system the designers would propose the initial state of the world and the desired goal, and the tool would provide all the possibilities to achieve that goal.

Chapter 6

Self-Validated Behaviour Trees through Reflective Components

In this chapter we are proposing a concrete *reflective component based system* (Llansó et al., 2009), which is used to validate Behaviour Trees over a concrete entity prior to the execution of the tree. In the process of validating Behaviour Trees, the different branches of the tree are covered. Their leaves (final nodes which contains an action) are checked with the entity, looking for possible failures caused by intrinsic limitations of the entity. Then, depending on the tool that has been proposed, failures would be reported, fixed or ignored.

The rest of the chapter runs as follows. The next section (Section 6.1) will present the typical kinds of failures that can be found in Behaviour Trees and how we propose to extend these kinds of failures. After that, we will propose a possible implementation for the *Reflective Components* of this proposal (Section 6.2). Firstly we will explain a coarse-grained approach an then will be explain the fine-grained approach with a detailed pseudo-code implementation. Next, we will explain the implementation part corresponding to the extended behaviour trees that we introduce (Section 6.3). To complete the proposal, the section 6.4 will expose a detailed example of how the system would work and how some specific component could be implemented in order to work properly. Finally, the different uses that the system would have will be tackled in Section 6.5. This section is split in two parts. The first one will present the uses during the design time whilst the second one will present the uses during the execution of the game.

6.1 Failures in Behaviour Trees caused by Intrinsic Nature of the Entity

As we have shown earlier, a behaviour tree describes the actions that an avatar should execute within the environment using a tree hierarchy. Behaviour trees are then assigned to concrete entities either in design time or once the game is being executed.

During the game's execution, some failures can arise due to these assignments and due to the state of the world. These failures can be due to several reasons and they all are failures related to the basic actions. Most of the implementations of behaviour trees

distinguish between two different kinds of failures of the basic actions:

- Failures prior to the execution: these errors appear *before* the beginning of the execution of the action. The action ends without having made any changes to the world so usually, the solution involves local replanning. A typical example of this is the failure of a `Move-To` action because there is no path to the target position. In this case, the replanning usually consist of finding a new path in order to arrive at the same target position.

- Failures during execution: before the execution, the action checks the environment and it establishes that the action is able to completely perform the task. However, if something changed in the virtual environment the task would not be able to be completed (i.e. these errors are due to the ever changing nature of the world). When these failures happen the action has already made changes to the virtual environment. When these failures take place, the response usually looks for another alternative in a different branch of the tree that tries to achieve the same goal. One example is the failure of a `Move-To` action because the path is suddenly blocked, and there is no other way to reach the target.

These failures are due to the state of the environment and they are not due to the abilities of the entity, which is associated with the behaviour tree. However we can extend the kinds of errors we can find in a behaviour tree. Generally speaking, a behaviour tree fails when its actions cannot be executed (we assume that the failure of composite or internal nodes depend on the success or failure of their children that, in the end, are basic actions). Therefore, it makes sense to carefully analyse new conditions that may cause an action to fail:

- The NPC assigned to the behaviour tree does not have the ability to execute the action. An example of this is an action that follows a path within the environment but requires the NPC to fly. If the NPC does not have that ability, the action will fail. This is an intrinsic limitation of the entity, not a failure of the action itself as a result of the environment (such as in previous examples).

- The NPC has the ability to execute the action but it cannot do the action under the concrete conditions created by the designer. This is related to the *parameters* of the actions rather than it's nature. For example, an entity may be able to carry objects but it might not be able to carry a specific object because it was too heavy. Once again, the failure is due to an inherent limitation of the NPC.

These failures are due to assignations of behaviour trees to entities. So, to summarize, we can say that there are two kinds of errors: those due to the intrinsic nature of the entity that will try to execute the actions (the extended kinds of failures), and those that are related to the state of the environment (the previous ones).

The system, proposed in this chapter, focuses on the extended kind of errors due to the intrinsic nature of the entity that tries to execute the Behaviour tree: Those errors

related to the abstract abilities of the entity (can it fly or carry objects?) and those limitations of the entity while performing the ability itself (maximum flight distance or maximum weight, of an object, that can be carried).

6.2 Implementation of Extended Reflective Components and Entity

As it has been said in section 5.2, the implementation of the process of validating behaviour trees is based on the components. If the entity is specified in terms of its components, and that a component can be seen as an *ability* that the entity has, it makes sense therefore to try to identify the failures related to the inherent nature of the entities using such a description. In order to do that, we will see, as a starting point, the explicit knowledge that is available both in the list of components and in the Behaviour Trees, which contains the list of actions needed to execute the behaviour.

6.2.1 Coarse-Grained Approach

According to the classification of failures as listed in section 6.1, the implementation has to cope with two different limitations: those inherent to the actions and those related to the parameters given to the actions.

For the first ones, and taking into account that every action is performed by a component, the easy (and naive) approach is to make direct associations between actions in final nodes of behaviour trees and components which are capable of executing these actions.

As an example, let us suppose a behaviour tree that defined a character behaviour that consisted of patrolling from one point to another and, when another character was perceived, it shot it with a gun. The final behaviour tree would have (together with composite nodes and the condition node related to the perception of another entity) a `Move-To` and an `Attack` action. The test would be to check if the entity assigned to the behaviour tree had a `Move-To` component for the patrol and an `Attack` component for shooting at the enemy.

To implement this idea it is enough to have a table of pairs, in which each pair is made up of a behaviour tree action and a list of components that carries out the abovementioned action. So, to validate a behaviour tree assignation would suffice to check that, for each action, the entity has at least one component of the component list associated with each action in the table.

Unfortunately, though the idea is easy to implement, it is not precise enough, because it gives the designer false positives. In more specific cases, the entity would not be able to perform the behaviour whilst our implementation would assure that it would.

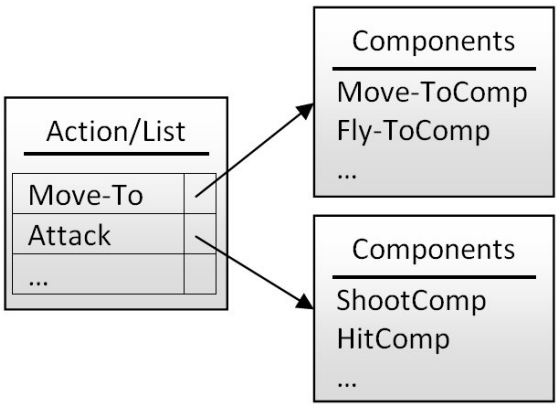

Figure 6.1: Table scheme with actions and lists of components that are able to carry out these actions.

6.2.2 A Fine-Grained Approach

There are two different reasons why the entity would not be able to perform a behaviour whilst our previous implementation would assure that it would. Some components could be associated with behaviour tree actions, but they could not always be able to carry out all these kinds of actions by themselves, either because they need the collaboration of other components, which may not be in the entity, or because the component cannot correctly execute the action with the parameters associated with this action.

For example, let us suppose a behaviour tree that had an action that must make an entity fly from one point to another. The entity may have a `Move-To` component, but the `AnimatedGraphic` component of the same entity would not be able to play the flight animation. When the `Move-To` component broadcasts the `ChangeAnimation` message with the "`flight`" parameter, no component would be able to perform the action, and therefore the behaviour tree would not be suitable for the entity which it wants to be associated with.

On the other hand, there can be a component that, let us say, allows the entity to take other objects. When the component is created, it takes some information from the map and archetype files (Section 3.2.2), such us the maximum weight that the entity can carry. If the AI of the entity decides to take an object, it will send a `Take` message with a parameter specifying the object to be taken. If the behaviour tree sent a message with an object whose weight exceeded the maximum load that the entity may carry, the message would not be successfully handled by any component.

In order to manage both kinds of errors and with the purpose of giving a fine-grained approach, we extend the `IComponent` interface, so that the components are able to be asked about their abilities for performing actions in the behaviour tree. Those question are made by asking the components if they are able to handle a concrete message according to their configuration. So, we shall query them using the same messages that behaviour tree actions will generate during the game execution to give instructions to

the entity. These messages include specific parameters of behaviour tree actions. Considering the previous examples, this new method would be used to check if there were some components capable of processing a `Go-To` message that requires the entity to fly, or a `Take` message with a concrete object. both particularities (requires the entity to fly or to take a concrete object) should come as parameters of the message.

As a starting point, it can be supposed that the proposed implementation consists of when a behaviour tree is assigned to an entity during the design time, the final behaviour tree actions would be gone through, one by one, asking the entity. An action would ask the entity through the `canEntityCarryOut()` method, if it (the action) could be executed with its parameters. The method would ask the components sequentially, through the `canComponentCarryOut()` method, about their abilities in executing this action, until a component returns true or the list has ended.

The specific components would be those responsible for implementing the `canComponentCarryOut()` method, declared in the interface of the component, reporting which actions could be performed. Depending on the component, it would automatically return true/false or it would check if their attributes allowed the action to be executed (for example comparing the maximum weight that the component can take with the weight of the object that is supposed to be taken).

More complex actions would require the component to recursively consult other components about their abilities to execute primitive actions such as pick a concrete animation. So in the previous example, the `Move-To` component would ask the entity components about if they had the ability to play the flight animation and it would return true/false depending of the answer of the other components.

It is obvious that components have to be initialized to receive messages and to check their abilities but, as has been explained in section 5.2, full initializations are not necessary. So every specific component would be responsible for implementing the `spawn()` method, declared in the `IComponent` interface. For example, during the game execution, a `Physic` component will create a physic entity, into the physic engine, that allows the entity to collide, to be moved by forces, etc. However it is not necessary during design time and the component might only need to know if the physic entity is static, dynamic or kinematic (to know which kinds of movements it is able to do) and to know if it is solid or if it is not (to know if other entities can collide with it or if they cannot).

Code Block 3 shows the pseudo-code of the implementation at this point.

```
class IComponent {
    ...
    virtual void spawn(bool full);
    virtual bool canComponentCarryOut(Message m) {
        return false;
    }
    ...
};
```

```
class Entity {
    List[IComponent] _components;
    ...
    bool canEntityCarryOut(Message m) {
        for each IComponent c in _components {
            if (c.canComponentCarryOut(m))
                return true;
        }
        return false;
    }
    ...
};

bool check(Entity e, BT bt) {
    for each Action a in bt.actions {
        Message m = a.getMsg();
        if (!e.canEntityCarryOut(m))
            return false;
    }
    return true;
}
```

Code Block 3: Pseudo-code of the implementation

6.3 Implementation of Extended Behaviour Trees

As has been said before, a starting point, in order to validate associations between behaviour trees and entities, could be to iterate over the list of actions of the behaviour tree, and if all the actions were able to be carried out by the entity the association would be validated. The problem with this is that this process is quite coarse-grain and its precision is not good enough. In this way, the system would only validate or invalidate associations, but if the system invalidated an association, it would not locate where and why this association was invalidated.

Therefore, a fine-grained approach should locate which branches of the behaviour tree were not able to be carried out by the entity and which the node and the reason that made it crash. Bearing in mind that a behaviour tree may have different internal nodes (Chapter 4), all these kinds of nodes have to be evaluated by different methods.

As its name denotes, a sequential node represents a chain of behaviours. Thus to validate a sequential node all its children nodes must have been validated before. Therefore, if there was one node of the sequence that was not validated, the sequential node would be invalidated knowing why and where the problem would be.

A parallel node could be analysed in the same way as the sequential nodes were analysed. As in sequential nodes, to validate a parallel node all its children nodes must have been validated before. The only difference between both nodes is that in a sequential node, its children have to be executed one by one whilst in a parallel node all its children have to be executed at the same time.

Both kinds of selector nodes (static and dynamic priority list) represent a behaviour that chooses between different ways of resolving a problem. So only one of the child nodes would be executed during the game rather than in previous examples, in which all the children would be executed. As a result of this, a fault detected in a child of the selector nodes was less critical than faults detected in a child of sequential and parallel nodes. This is because there were probably another choice (another child) selectable by the selector node. Therefore we could call these faults as warnings, instead of failures, if the child node that fails has at least one other brother node that has been validated.

So, to summarize, the method would be applied to the root node of the behaviour tree, then the method would be applied to its children and recursively to all the nodes of the tree. Finally, the leaves of the tree, which contains the final actions, would be checked and validated/invalidated.

As a result of this, failures and warnings would be located and associated with one branch of the behaviour tree. Therefore, how these failures and warnings would be fixed or reported, would depend on how the tool works. Nevertheless, the easier way for solving a warning is to remove the whole branch whilst failures must be treated with more care.

In this fine-grained approach, the implementation code has to be extended as in Code Block 4. `IComponent` and `Entity` classes would continue being the same as in Code Block 3. Nevertheless the classes of the nodes of the behaviour trees should be extended. This is because the check must be done recursively from the root of the tree to its leaves.

So, the `INode` interface would have a new virtual method, `check()`, which should be implemented by any class that inherits from the `INode` interface. This method would return two lists with failures and warnings. Each failure, or warning, would be in turn another list that would have the list of nodes involved in the failure, or warning. The first node would be the beginning of the failed branch and the last one is the leaf (final action) that made it crash.

In the `Action` (node) class, the `check()` method would ask the entity using the `canEntityCarryOut()` method, if the action could be executed with its parameters. This question would be done by sending the same message that was generated during the game execution and, as has been explained in section 6.2.1, the entity would broadcast the message to all of its components. If the entity returned false it would mean that the action cannot be carried out by this entity. Therefore this would be a failure and, as such, it shall be inserted into the failure list.

In the `Sequential` and the `Parallel` classes (or maybe in a common interface), the `check()` method would ask recursively the `check()` methods of the children nodes located in a list that these classes should have. The warning lists of every child would be merged to create the warning list of the `Sequential` or the `Parallel` node. In the same way, the

failure list would be created from the failure lists of the children but with the difference that it (the `Sequential` or the `Parallel` node) should be included into all the failures reported by the children. The node should include itself because, as has been explained before, if one node of the sequence (or of the parallelization) failed, the `Sequential` or the `Parallel` node would fail too.

The `check()` method of the `Selector` class would also ask recursively the `check()` method of its children but, in this case, we have to remember that a failure produced in a child node would be only a warning in a `Selector` node. So, the new warning list would be the merging of all the failures reported by the children and also the warnings reported by them. A `Selector` node would only trigger a failure if all their children failed. Thus in this case, the `Selector` node would include itself into the failure list that it reported.

Lastly, the global method that checked the association between behaviour trees and concrete entities would only call the `check()` method of the root of the tree and, as has been seen previously, all the nodes would be asked recursively.

```
typedef List[List[node]] TListErrors;

class INode {
    ...
    virtual {TListError,TListError} check(Entity e);
    ...
};

class Action : INode {
    ...
    {TListError,TListError} check(Entity e) {
        TListError failures;
        TListError warnings;
        Message m = this.getMsg();
        if (!e.canEntityCarryOut(m))
            failures.add([this]);
        return {failures,warnings};
    }
    ...
};

class ISequential&Parallel : INode {
    List[INode] _children;
    ...
    {TListError,TListError} check(Entity e) {
        TListError failures;
        TListError warnings;
        for each INode n in _children {
```

```
            TListError f_children;
            TListError w_children;
            {f_children,w_children} = n.check(e);
            //if a child fails, sequential node fails
            //it should be included into the failures
            for each List l in f_children {
                l.add(this);
            }
            failures.merge(f_children);
            warnings.merge(w_children);
        }
        return {failures,warnings};
    }
    ...
};

class Selector : INode {
    List[INode] _children;
    ...
    {TListError,TListError} check(Entity e) {
        TListError failures;
        TListError warnings;
        bool validated = false;
        for each INode n in _children {
            TListError f_children;
            TListError w_children;
            {f_children,w_children} = n.check(e);
            if(f_children.empty())
                validated = true;
            else {
                //if a child failed in a selector it would
                //be a warning instead of a failure
                warnings.merge(f_children);
            }
            warnings.merge(w_children);
        }
        //if no child is validated this node should
        //be into the failures
        if(!validated) {
            failures.add([this]);
        }
        return {failures,warnings};
    }
```

```
    ...
};

{TListError,TListError} check(Entity e, BT bt) {
    return bt.root().check(e);
}
```

Code Block 4: Pseudo-code of the implementation

6.4 Example

Let's suppose that we were creating a game where avatars would need coal for their subsistence. Designers would create behaviour trees such as the one shown in Figure 6.3 that represents a behaviour of "coal harvesting". It is compound by a composite node that executes in sequential order the action of `Look for a thing`, to look for a coal mine, and the selector node `Go and leave resources`. This selector node has two choices, which are represented by two different sequences of actions. In the first sequence, the avatar would find and would get in an excavator shovel, it would use it in order to go to the coal mine, it would load coal, it would drive to camp and it would unload the coal there. On the other hand, in the less priority sequence, the entity would move to the coal mine, it would find a piece of coal, then it would take it, it would return to the camp and finally it would leave the piece of coal there.

Once the tree was described, if it was associated with the entity described in Figure 6.2, the proposed system would validate or invalidate this association. To do it, the system would work, as has been described in Figure4, by invoking the `check()` method of the root node of the behaviour tree. Then, the root (`Coal harvesting` node) would call the `check()` methods of its children because it is a `Sequential` node. The `Sequential` node would receive the failures and warnings of its children and then it would propagate all of them adding itself, previously, to all the failures.

Nevertheless, to receive these failures and warnings, the `check()` methods of its children must be processed. The first node of the sequence (the `Look for a thing` node) is a final action and consequently, its `check()` method would call the `canEntityCarryOut()` method of the entity, with the message that would be produced during the execution of the tree as a parameter. If this method returned false, the `Look for a thing` node would be added as a new failure to the failure list but in this case, the action would be handled by the `Look-For` component so the `canEntityCarryOut()` method would return true.

The second and last node of the sequence of the root (the `Go and leave resources` node) is a `Selector` node with two different choices that are two `Sequential` nodes. Therefore, the `Go and leave resources` node would ask its children using their `check()` methods.

```
<BluePrints>
    ...
    <entity type = "Labourer">
        <comp type = "AnimatedGraphic"/>
        <comp type = "Physic"/>
        <comp type = "BTExecuter"/>
        <comp type = "Move-To"/>
        <comp type = "DummyBehaviours"/>
        <comp type = "Look-For"/>
        <comp type = "Take"/>
        <comp type = "Skills"/>
        <comp type = "Use-Vehicle"/>
    </entity>
    ...
</BluePrints>
```

a) Blueprints file

```
<archetypes>
    ...
    <entity type = "Labourer">
        <attrib
            name = "static"
            value = "false"/>
        <attrib
            name = "solid"
            value = "true"/>
        <attrib
            name = "kinematic"
            value = "true"/>
        <attrib
            name = "model"
            value = "labourer.n2"/>
        <attrib
            name = "strength"
            value = "strong"/>
        ...
    </entity>
    ...
</archetypes>
```

b) Archetypes file

Figure 6.2: `Labourer` entity built by components.

When these different choices were asked, as they are sequences, they would ask, in turn, the `check()` methods of their children. All the children of both sequences are final `Actions`, so when they were checked all of them would call the `canEntityCarryOut()` method of the entity, with the messages that they would produce during the execution.

At this point, only the different final actions were unprocessed. `Move-To` actions would be handled by the `Move-To` component. When this component was consulted through the `canComponentCarryOut()` method, it would, in turn, ask through the `canEntityCarryOut()` method consulting if the entity were able to play a *walk* animation. The `AnimatedGraphic` component would be able to carry out the animation because the associated model (*labourer.n2*) has it.

The `Look for a thing` actions would be carried out by the `Look-For` component as has been mentioned before. The `Take` action and the `Leave` action would be carried out by the `Take` component. As the specific piece of coal that the entity would take is not hard-coded, it would not be known during the validation. Because of this the `Take` component would not have to check extra conditions such as the entity having enough strength to take the weight of the specific piece of coal.

The `Get in vehicle` action, the `Get out vehicle` action and the `Drive-To` ac-

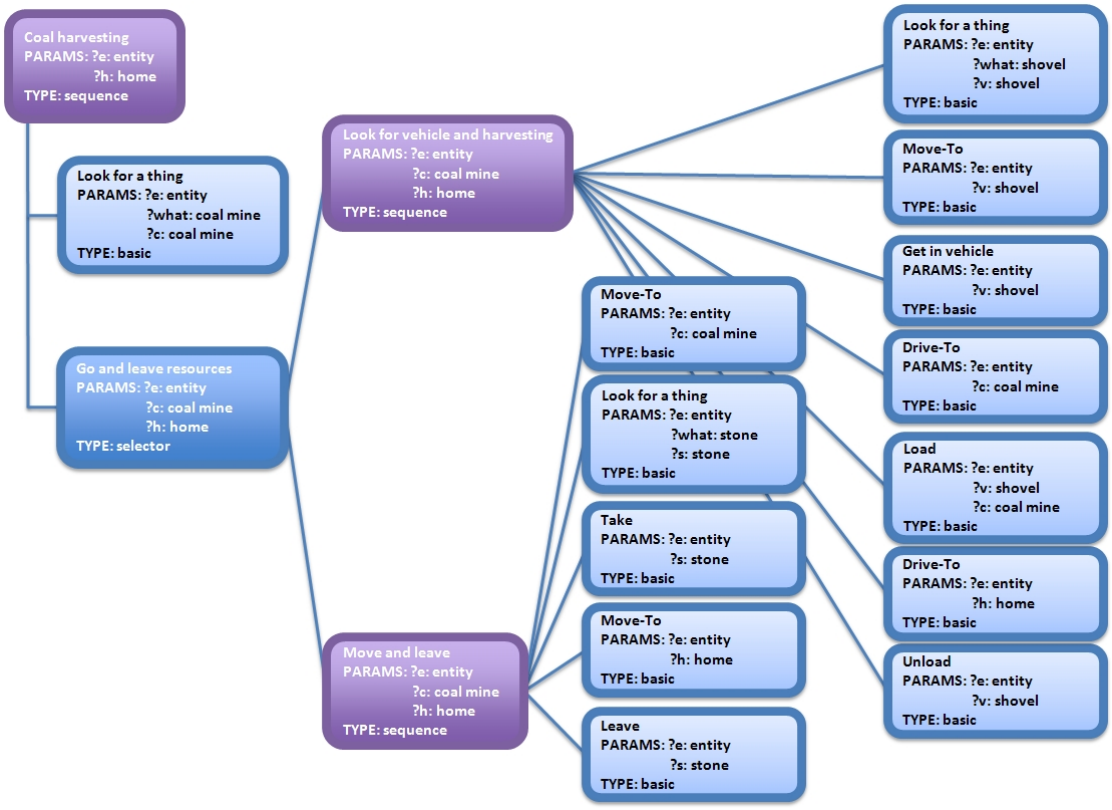

Figure 6.3: Behaviour Tree 1 of "coal harvesting" type.

tion would be carried out by the `Use-Vehicle` component. When this component were consulted, by these different actions, through the `canComponentCarryOut()` method, it would, in turn, ask through the `canEntityCarryOut()` method consulting if the entity would be able to play those corresponding animations.

It could be thought that `Load` action and `Unload` action would be carried out by the `Use-Vehicle` component too, but let us imagine that this component was implemented to cover the basic actions that could be done with a vehicle. So, to execute these actions, the entity should have another specific component to use an excavator shovel. Due to this component not being specified in the blueprints file for the `Labourer` entity, these actions would fail and both actions would report a failure to their father, the `Look for vehicle and harvesting Sequence` node. This node would add itself to the failures and it would propagate these failures to the `Go and leave resources` node. This node would receive these failures and, because of it would be a `Selector` node, these failures would be turned into warnings. Finally, these warnings would be reported to the root node and, in turn, the system would finish by receiving these two warnings and no failures.

in Code Block 5 can be seen how `canComponentCarryOut()` methods of some components could be implemented. As an example, let us expose how a `Take` action, with

a hard-coded object to take, was checked. The action would ask the entity using the `canEntityCarryOut()` method and passing the message with the object which was to be taken. The entity would ask its components by passing the same message. The `Take` component, during its `canComponentCarryOut()` method, would create and would ask the entity with two different messages in order to know if the entity was able to play the *take* animation and if it was able to produce enough force to take the weight of the concrete object. If both queries returned true or if there was not a concrete object to take and the entity had the animation, the method would return true too. In any other case, the entity would return false.

The `canComponentCarryOut()` method of the `AnimatedGraphic` component would return true if its model had the animation or if the name of the animation was not specified. Meanwhile, The `canComponentCarryOut()` method of the `Skills` component would return true if its force was greater than or equal to the weight of the object (if there was a concrete object specified).

```
class Take : IComponent {
    ...
    bool canComponentCarryOut(Message m) {
        if(IS_INSTANCE_OF(m,TakeMsg)) {
            if(m.getObject() != NULL) {
                Message ef= new ExertForceMsg(m.getObject().getWeight());
                Message sa= new SetAnimationMsg(‘‘take’’);
                return _entity.canEntityCarryOut(ef) &&
                       _entity.canEntityCarryOut(as);
            }
            Message sa= new SetAnimationMsg(‘‘take’’);
            return _entity.canEntityCarryOut(as);
        }
        return false;
    }
    ...
};

class AnimatedGraphic : Graphic {
    Model _model;
    ...
    bool canComponentCarryOut(Message m) {
        if(Graphic::canComponentCarryOut(m))
            return true;
        if(IS_INSTANCE_OF(m,SetAnimationMsg)) {
            if(m.getAnimation != NULL)
                return _model.hasAnimation(m.getAnimation());
            return true;
```

```
        }
        return false;
    }
    ...
};

class Skills : IComponent {
    int _force;
    ...
    bool canComponentCarryOut(Message m) {
        ...
        if(IS_INSTANCE_OF(m,ExertForceMsg)) {
            if(m.getWeight() != NULL) {
                return _force >= m.getWeight();
            }
            return true;
        }
        return false;
    }
    ...
};
```

Code Block 5:
Pseudo-code of the `canComponentCarryOut()` methods of some components

6.5 Different Uses

The *reflective component based system* that we are proposing can be used in different ways: During the design time and during the execution of the game.

During the design time, designers would use *reflective components* to check direct associations that they had made between entities and behaviour trees in order to verify these links. It would give designers the chance of fixing many wrong associations without the need to debug the game.

During the execution of the game, the game itself would use *reflective components* to check if the links, made during the design time, could be carried out under the current circumstances of the environment. Although the associations were checked during the design time, some special circumstances of the environment may prevent the entity in carrying out some validated actions of the behaviour tree. This use would give the game engine the chance of replanning the carrying out of goals when some branches of the behaviour tree had not been carried out under special circumstances of the environment.

6.5.1 During the Design Time

During the design time, the system would try to identify the failures due to the intrinsic nature of the entity in order to check if there were some guarantees, about the success of the execution of the behaviour trees, as soon as possible. With this technique, designers would have an extra check that would assist them when creating behaviour trees and NPCs. Due to the check would be done *before* the execution of the behaviour tree, designers would be more confident about the correct link between NPCs and Behaviour trees.

Let us imagine a tool that would use the concrete *reflective component based system*, which we have proposed in this chapter, in order to check associations between entities and behaviour trees during the design time. This tool could check these associations, and use the failure and warning lists to help the designer in the process of fixing the tree (or the entity).

The tool would provide a graphic interface that would present the tree. When the tool processed failures and warnings, the affected branches would be stressed and for each failed branch, the system would offer some different choices to fix the branch. For example, the tool might find another way to achieve the same goal that the failed branch tried to achieve. In the next chapter we will propose another *reflective component based system* that would help the tool to find different ways to achieve a goal. Other choice would be to use case-based reasoning (CBR) in order to retrieve branches, of other designed behaviour trees, that achieve the same goal. Another way for fixing a failure could be to offer a list of components that would allow the entity to carry out the action that the entity was not able to carry out before.

In the same way, the warnings would be stressed and some choices could be offered in order to fix them but with the difference that fixing them would not be necessary. This is because if the designer did not correct a warning, the tool would rule out the warned branch and the resultant tree would be accepted.

Of course, the tool would let designers fix the branch manually by adding a new branch instead of the failed one and also, the tool would let designers recheck associations whenever they wanted.

To show how it would work, let us propose an example of a complex behaviour tree (Figure 6.4) that would fail in its association with the entity shown in Figure 6.5. The behaviour tree represents different alternatives of wood harvesting. in Figure 6.4 the warnings (orange branches with dashed borders) and failures (red branches with dotted borders) can be seen, reported in the association of the entity with the behaviour tree. These faults would arise according to a lack of a specific component that was able to carry out the *Look for a thing* behaviour. Let us remember that the actions that fail into a `Selector` node are catalogued as warnings, instead of failures, because there could be other choices.

At this point, the tool would show the failures and warnings, in a similar way as in Figure, and it would propose alternatives to fix them. The easier alternative would be to add a new component to the entity such as a `Look-For` component, which was able

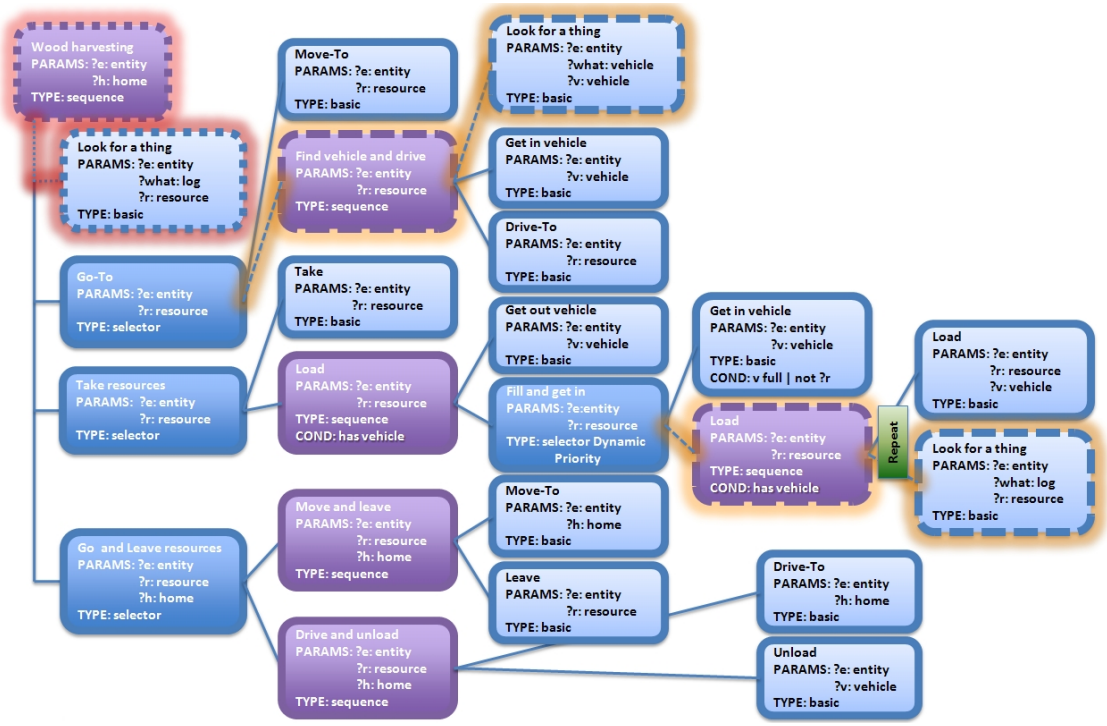

Figure 6.4: Behaviour Tree of go sounding behaviour.

to execute the `Look for a thing` action, and, in this way, the failed branches could be carried out. The other alternatives would be related to the modification of the behaviour tree. This would be done through the use of a planner or by retrieving branches of other behaviour trees previously designed (CBR).

Both methods could offer the alternative shown in Figure 6.6. This behaviour tree fixes the problem with a collection of nodes that would make the entity wander until it saw another character. When another character is perceived, the behaviour tree would move the entity to the character and would make the entity ask about the thing that it was looking for. The behaviour would finish when the thing, which was looked for, was found. By hard-coding the thing that was desired, this behaviour could be used to look for both a log and a vehicle.

Once the tool is offered the different fixes, the designer was the person responsible for choosing between the different alternatives, or of creating new valid branches in order to replace the failed ones.

6.5.2 During the Execution of the Game

As during the design time, during the execution of the game the system would try to identify the failures and warnings due to the intrinsic nature of the entity before the execution of the behaviour. In this case, the faults could arise because of two different

```
<BluePrints>
    ...
    <entity type = "Human">
        <comp type = "AnimatedGraphic"/>
        <comp type = "Physic"/>
        <comp type = "BTExecuter"/>
        <comp type = "Move-To"/>
        <comp type = "DummyBehaviours"/>
        <comp type = "Take"/>
        <comp type = "Skills"/>
        <comp type = "Use-Vehicle"/>
        <comp type = "Communicate"/>
    </entity>
    ...
</BluePrints>
```

a) Blueprints file

```
<archetypes>
    ...
    <entity type = "Human">
        <attrib
            name = "static"
            value = "false"/>
        <attrib
            name = "solid"
            value = "true"/>
        <attrib
            name = "kinematic"
            value = "true"/>
        <attrib
            name = "model"
            value = "human.n2"/>
        <attrib
            name = "strength"
            value = "strong"/>
        ...
    </entity>
    ...
</archetypes>
```

b) Archetypes file

Figure 6.5: Partial list of blueprints file

reasons. The obvious one would be because the associations between the entity and the behaviour tree were not checked during the design time. The other possible reason could be that, during the design time, there were some unknown things and because of this, a whole check was not possible.

As an example of the second possibility, let us suppose that a behaviour tree of "wood harvesting" that made the entity find a new piece of wood, to take it and to bring it home. Also, let us suppose that the process of finding the piece of wood to be taken was made by another complex behaviour that had been previously checked. Thus, the logs to be taken were not *hard-coded* in the tree. So if the entity had had the abilities of taking and bringing logs, the validation between it and the behaviour tree would have been made during the design time. Nevertheless, the weight of the logs that the entity would have to take was not known during the design time because these logs would be selected by the other behaviour during the execution of the game. Therefore if during the execution of the game the behaviour, which decided the log which was to be taken, chose a too heavy log that the entity would not be able to take, the check between the "wood harvesting" behaviour tree and the entity would have a failure although it was validated at the design time.

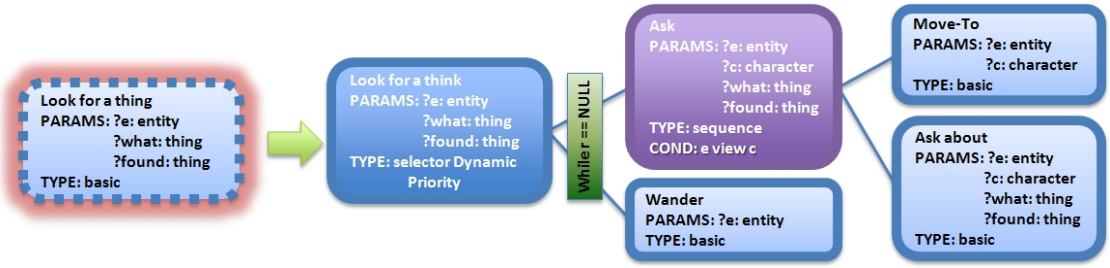

Figure 6.6: Behaviour Tree of go sounding behaviour.

Now failures and warnings can arise during the execution of the game, consequently the game would have to be ready to fix them. A warning would be easily fixed by removing the whole warned branch. This would be the best choice because warnings only arise into `Selector` nodes, so other choices would be expected by the choice that the `Selector` node would have to make. Remembering that if there were not any other choices, a failure would arise and this failure would be fixed.

On the other hand, a failure would be more difficult to fix because the intention of the designer would be not known at that point. The designers are those responsible for design of both entities and behaviour trees. Consequently, during the execution of the game, the game engine must be as faithful as it can to design entities and trees.

Nevertheless when the game was notified about a failure, it would not know if the failure would have been due to either the entity, which could have a component missing, or the tree, which could have invalid actions. For example, if during the execution of the game a failure was reported, because of there were an entity that was not able to fly associated with a behaviour tree that had an action that made the entity fly from one point to another, which would be the one responsible of this failure?. Could be supposed that behaviour trees would be always accomplished, as they were built, because if they would not, the story plot would be broken. So in this case, the failure would be fixed by allowing the entity to do actions far from its possibilities. Although also could be supposed that the entities were always well built and if a behaviour tree reported a failure, the game should try to fix it by accomplishing the same goals, which the behaviour tree tried to accomplish, in another way.

As it would be impossible to know which would be the intention of the designer at the execution of the game, during the conception of the game, a decision has to be taken about which approximation should have a higher priority.

If the decision was to think that the behaviour trees would always be accomplished as they were built, when a failure was reported, the game engine would try to fix it by changing the entity associated with the behaviour tree that reported the failure. Let us suppose that a simple example with an entity that represents a stone and a behaviour tree that moves the stone from one point to another. The entity, as can be seen in Figure 6.7, would have a `BTExecuter` component, a `Physic` component, a `Graphic` component and some associated parameters such as `static` parameter that the `Physic` component would

take in order to represent the stone into the physic engine as a static and immovable entity, or the `model` parameter that the `Graphic` component would take in order that the rock would be rendered by the graphic engine. The behaviour tree would be like the one in Figure 6.8, which launch, in parallel, both a movement and a sound of the movement.

```
<blueprints>
    ...
    <entity type = "Stone">
        <comp type = "BTExecuter"/>
        <comp type = "Graphic"/>
        <comp type = "Physic"/>
    </entity>
    ...
</blueprints>
```

a) Blueprints file

```
<archetypes>
    ...
    <entity type = "Stone">
        <attrib
            name = "static"
            value = "true"/>
        <attrib
            name = "solid"
            value = "true"/>
        <attrib
            name = "model"
            value = "stone.n2"/>
    </entity>
    ...
</archetypes>
```

b) Archetypes file

Figure 6.7: Partial list of blueprints file

The problem with the association of the entity with the behaviour tree would be that the entity would not be able to move itself nor to emit a sound. This problem would arise according to a lack of specific components. Therefore, in this case, both a `Move-To` component and a `Sound` component could be added to the entity during the execution of the behaviour tree. In this way, the entity would be able to carry out the two actions and the game engine would be faithful to the designed behaviour tree.

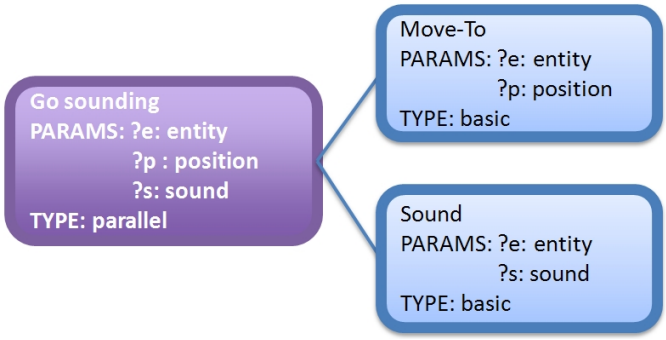

Figure 6.8: Behaviour Tree of go sounding behaviour.

Otherwise, if the taken decision was to think that the entities are always well built, and they are only able to carry out the actions that their components allow them, when a failure arose, the game engine would try to fix it by changing the behaviour tree, instead of the entity that was associated with. For this case let us suppose the example in which there were an human entity and a behaviour tree that made the entity to fly in order to take an object. The entity, as has been described before in Figure 6.5, would have some components and some associated parameters whilst the behaviour tree, as can be seen in Figure 6.9, would be made up of a `Sequence` node with two basic actions: `Fly-To` and `Take`.

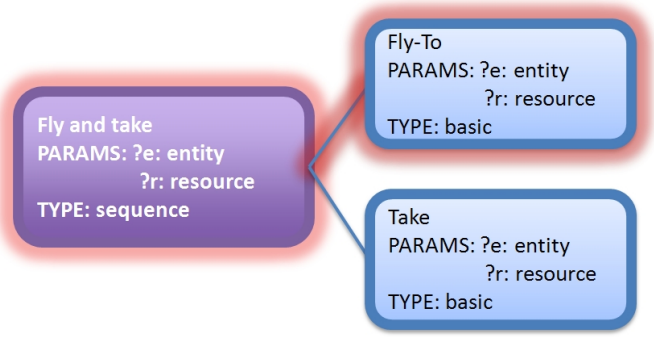

Figure 6.9: Behaviour Tree of to fly and to take.

In this case, the main problem of the association of the entity with the behaviour tree, would be that the human entity would not be able to fly. This would be because of that the `Move-To` component does not allow flight movements and the entity had not got any other component that let it fly. Furthermore there could be another problem. it would be that the entity was not able to take the specific resource, which is hard-coded into the behaviour tree, due to the resource was too heavy for the `strength` of the entity, which was described in the blueprints file (Figure 6.5). However, this example will be only focused on the first problem.

The failure reported to the game would be caused by the `Fly-To` action. So, to solve it, the game engine would have to replanning the carrying out of the goal that the `Fly-To` action pursued. Consequently, one solution, as has been seen in Section 6.5.1, could be to use the *reflective component based system* that we will propose in the next chapter. This system would try to find different ways to achieve a specified goal helped by a planner. In this way, the planner could find a sequence of actions that replaced the `Fly-To` action. The goal of the `Fly-To` action is to move the entity from one point to another. So, if the `Fly-To` action was a flight movement that ended in the top of a mountain and if the entity was not able to fly, the planner could infer an equivalent sequence of actions that were to move at the bottom of the mountain and then to climb to the top. The resultant behaviour tree would look like the one shown in Figure 6.10, where the new branch is stressed.

Another way to fix the failure could be to use case-base reasoning (CBR). If the system

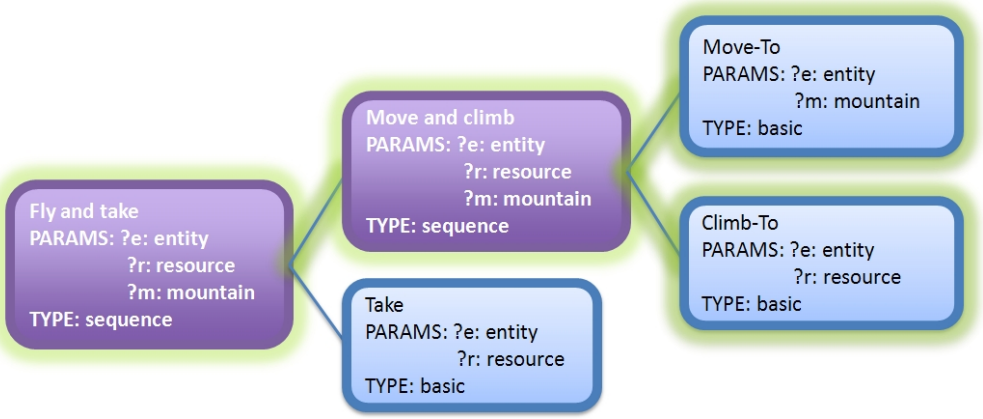

Figure 6.10: Behaviour Tree of to move, to climb and to take.

was provided with a base of cases, which had some different cases/behaviours that achieve different goals, when a failure arose, the system would look for the case/behaviour that would be more similar to the behaviour that triggered the failure and that could be carried out by the entity. To know the similarity between two behaviours the system would have to have an ontology of behaviours. In this way the system could look for behaviours of the same ontology class or behaviours of ontology classes with the nearest ancestor. However, the system should have more factors to measure the similarity between two behaviours. This factors may be related with the state of the game (the level, the boredom of the player, etc.) or with the state of the entity (its life, its strength, etc.). Even during the execution of the game, the base of cases could be increased by adding new cases that the entity is carrying out.

However, these processes would not work properly every time. In the previous example of the *flight* behaviour, the solution of replacing the failed branches could not be good enough. This could happen when, for example, the mountain, in which the resource was located, was too steep in order to climb to the top and there was not any other choice offered by the planer nor by the base of cases. On the other hand, the solution of adding new components, in order to fix the failures, could not be valid either in some cases. Following the same example, a `Fly-To` component could be added to the entity and in this way it would be supposed that the failure would be fixed. Although probably, the `AnimatedGraphic` component, attached to the entity, would not have a *flight* animation. So the complete *flight* action would not be carried out properly.

Chapter 7

Generating New Behaviours by Means of Abstracted Plan Traces

The creation of behaviour trees is a difficult task that designers usually perform by means of a try-and-fail process. This process is usually manual and the designers must take into account the great amount of basic behaviours or actions and the different ways available to combine them. Moreover, the ultimate goal of the design of behaviour trees is to create trees that work in different scenarios to promote reusability and to cope with every different situation. Besides, developers, that implement the basic behaviours, and designers, that use them to build these behaviour trees, are usually different groups, so it is very important to have clear description for each behaviour to avoid misunderstandings. Unfortunately, the consequence is that the final quality of the behaviour trees depends to a large extent upon the ability and experience of designers.

Because of the previous reasons, we will specify a system that would be able to deal with behaviour trees by generating different solutions, to achieve goals, having into account the different scenarios in where the entity could stay. Our proposal would consist on the use of a combination of planning and ontologies. A planner would be able to suggest a set of plans that might be easily turn into behaviour trees. Then, the system could be adapted for design tools that helped designers in the task of creating behaviour trees, or the system could be added to the game engine for suggesting alternative behaviours to replace those that were not able of being carried out.

To develop this system by the traditional way, the information would be duplicated in the side of the planner and in the C++ classes programmed for the game. This duplicity could lead to errors, due to someone who implemented a new component might forget to replicate this new knowledge in the side of the planner. Furthermore, the team of people that worked in the planner could not be the same people who worked in the development of the C++ classes. This possibility would make the coordination more and more difficult. In order to fix this problem, we will expose a *reflective component based system* (Sánchez-Ruiz et al., 2009a,b) that would be specified to make the task of generating the information needed by the planner easier. In this proposal the reflective components, which would be used to define the different entities of the game, would be also those responsible of generating the planner information. In this way, all the information would be defined in only one place: the components.

The rest of the chapter runs as follow: In the next section, how to create behaviour trees, helped by planning with ontologies, will be explained. Then how to use the *reflective components* to improve the process will be shown. Following this section, an implementation approach will be developed and finally, the different uses of this system will be discussed with two different examples: during the

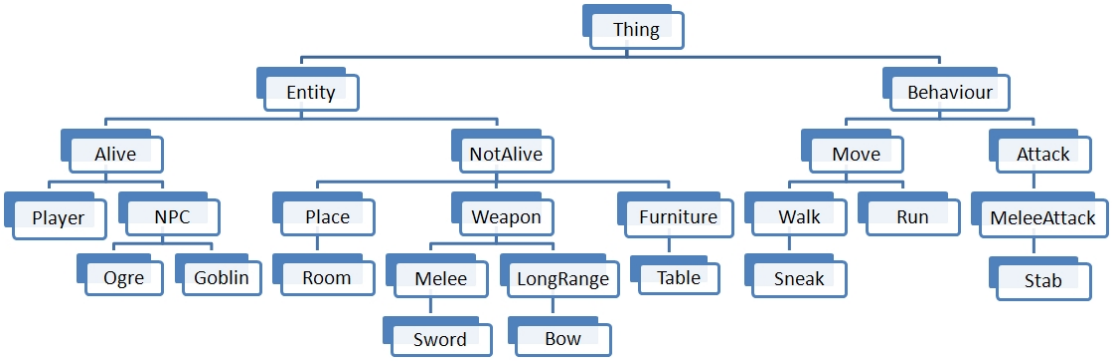

Figure 7.1: Ontology that defines the domain vocabulary

design time and during the execution of the game.

7.1 Planning with Ontologies to Support the Behaviour Tree Creation

In order to use planning, the domain and planning actions needed to be described by using a formal language. Unfortunately, this formalization process would require someone with experience in knowledge representation using symbolic languages, but hopefully the time spent in this task would be recover later during the creation of behaviour trees.

The description of a planning domain includes two main parts: (a) the description of the predicates, which conform the domain vocabulary, and how this predicates are related with each other, and (b) a set of planning actions. We propose the use of *ontologies* to represent the first part, that is, the vocabulary and the domain constraints. Ontologies are a standard mechanism for knowledge representation and sharing based on conceptual hierarchies defined using the *is-a* relation. Abstract concepts like `Entity` or `Behaviour` are located towards the top of the taxonomy, and specific concepts like `Ogre`, `Sword`, or `Stab` are classified towards the leaves of the taxonomies. Figure 7.1 shows an example of ontology that intuitively describes different components of a game. The set of planning actions, on the other hand, is strongly related to the available basic behaviours in the game. Planning actions are described in terms of preconditions and effects using the vocabulary available in the domain ontology.

Besides the primitive terminology, ontologies include logic axioms and descriptions that relate the different terms and allow certain reasoning capabilities. Reasoning is based on the formal representation of terms and descriptions using a knowledge representation language such as OWL-DL W3C Consortium (2004), an standard language based on Description Logics (DLs) Baader et al. (2003).

DLs are expressive subsets of First Order Logic with good reasoning properties that are specially designed to represent structured knowledge. DLs represent knowledge us-

ing *concepts*, *roles* and *individuals*. *Concepts* represent categories or types described by means of potential complex formulas, *roles* are binary relations, and *individuals* (conceptual instances) are entities that represent objects in the domain. For example, following Figure 7.1, *Room* is a concept defined as a subconcept of the concept *Place*, *inRoom* is a role that relates entities and rooms, and *inRoom(ent1,room1)* is an assertion that sets that the individual *ent1* is a type of *Entity*, *room1* is a type of *Room* and besides the entity *ent1* is located in the room *room1*.

The use of ontologies would allow us to infer general plans from the specific ones making them valid to be used in more cases than the concretes states of the world that were given to the planner. Because of this, this proposal would be based in the use of a planner called DLPlan[1], which would be able to generalize the resulting plans using the domain ontology. In this way, the inferred plans would not be only specific plans for the scenario given to the planner. it could be say that the planner would turn plans into general strategies that could be reused in a wide set of situations.

For example, if the planner determined that a NPC had to kill an enemy using a melee attack, a normal planner would generate as many different plans as melee weapons the NPC had. However, DLPlan was able to generalize those concrete plans and, in this way, it would infer just one plan in which the weapon parameter had the generic type *MeleeWeapon*. Later, in the example of Section 7.4.1.1 an extensive example will be presented by showing how useful this feature of DLPlan is.

Finally, the process by which these plans/strategies would be turned into behaviour trees could be manual or automatic depending on the tool that we had in mind. It is important to remark that, in design tools, automate this process would not be too important due to the fact that the planner would work with a limited model of the game, whilst the designers would take into account much more factors (like story plot or special situations) in order to select behaviours, to modify preconditions under certain circumstances and to set the priority of each alternative. In these kinds of tools, the inferred plans would be only suggestions that designers would modify, validate or reject.

On the other hand, to integrate this system into the game engine, the process by which inferred strategies would be turned into behaviour trees should be automatic in order to use them directly. It is obvious because during the execution of the game this process should be transparent to the user.

7.2 Generating the Planning Domain by Using Reflective Components

To be able to use planning techniques, a symbolic representation of the world would be needed as well as the actions that each type of entity can perform. The basic approach would be to create this description from scratch. However, this information would be, at least partially, already in the C++ classes, which programmers would have to implement in order to develop the game, and in the configuration files, which would define the

[1]Freely available at *http://sourceforge.net/projects/dlplan/*

different kinds of entities. Therefore, as it will be seen, this would lead to a not desirable duplication of information. In this section, our proposal to avoid it will be described making use of self-described software (reflective components) and files with enriched entity descriptions.

To summarize, the planner would need two different kind of information:

- A *symbolic* representation of the world. As it has been said, the vocabulary to describe the world would be defined by means of an ontology similar to the one shown in Figure 7.1.

- A set of operators that describe all the actions that can be carried out in the world. This operators should somehow codify which ontological entities would be able to carry out which actions over the environment.

On the implementation side, the code base would have a representation of every different entity type in the virtual environment. On the component approach, this usually comes specified in the form of the *blueprints* file and the *archetypes* file (Figure 7.2). All of this information would be equivalent to the ontology used by the planner.

As regards the operators, in order to allow an entity to be able to execute an action (which would have to be defined as an operator into the planner), this entity should be provided with a concrete skill. This is usually done by creating a new component and attaching it to the entity using the *blueprint* file. Once again, the information would be located, and some of it duplicated, in separated places: components and operators.

Therefore, both kind of knowledge that the planner needed, would be duplicated in the implementation side. This duplicity could lead to errors, due to the fact that someone who implemented a new component could forget to replicate this new knowledge into the *symbolic* representation. This could be worse in big projects because the team of people that works in the generation of the planning domain are not the same people that work in the developing of the game itself. In order to fix this problem, our proposal would consist in keeping the whole knowledge in the components. Developers would be those responsible for creating components that would be able to describe themselves in terms of the *symbolic* description that the planner would need to perform its task. So, in this way, the knowledge would be centralized in only one place, avoiding duplicates, mistakes and errors. As it will be seen later, every time that a new component was created, it would be automatically queried to regenerate the domain description that the planner would use.

In order to semi-automate the task of building the *symbolic* description that the planner would need, the process would start with a base domain ontology that could be seen as the basic vocabulary of the game genre, which might be independent of the concrete game that would being developed. This domain ontology would be taken for granted and it would include the basic vocabulary for describing the new type of entities and actions that the game would incorporate. In other words, the process would start with an ontology similar to the one in Figure 7.1 but without the leaves that

```
<blueprints>
    ...
    <entity type = "Ogre"
            ontType = "Ogre"
            parentOnt = "Monster">
        <comp type = "BTExecuter"/>
        <comp type = "Graphic"/>
        <comp type = "Physic"/>
        <comp type = "Take-Cover"/>
        <comp type = "Melee-Attack"/>
        <comp type = "Charge-At"/>
        <comp type = "Take"/>
        <comp type = "Move-To"/>
    </entity>
    <entity type = "Goblin"
            ontType = "Goblin"
            parentOnt = "Monster">
        <comp type = "BTExecuter"/>
        <comp type = "Graphic"/>
        <comp type = "Physic"/>
        <comp type = "Take-Cover"/>
        <comp type = "LongRange-Attack"/>
        <comp type = "Charge-At"/>
        <comp type = "Take"/>
        <comp type = "Move-To"/>
        <comp type = "Sneak-To"/>
    </entity>
    ...
</blueprints>
```

a) Blueprints file

```
<archetypes>
    ...
    <entity type = "Ogre">
        <attrib
            name = "strength"
            value = "strong"/>
        <attrib
            name = "weapon-tech"
            value = "rudimentary"/>
        <attrib
            name = "height"
            value = "tall"/>
    </entity>
    <entity type = "Goblin">
        <attrib
            name = "strength"
            value = "weak"/>
        <attrib
            name = "weapon-tech"
            value = "both"/>
        <attrib
            name = "height"
            value = "short"/>
    </entity>
    ...
</archetypes>
```

b) Archetypes file

Figure 7.2: Partial list of blueprints file

correspond to the concrete types of entities in the game. Then this base ontology would be automatically completed using the knowledge of the architecture of reflective components.

With the purpose of populating the ontology with the new entities, the system would use the *blueprints* file and the *archetypes* file. As it is shown in Chapter 3, these files would have an entry per game entity, describing the set of components and attributes that it had.

For example, Figure 7.2 shows two types of entities: the `Ogre` and the `Goblin`. These entities differ because of their different abilities, such as differences in their types of attack, and the parameters of these abilities. In this case, both units can perform `Melee` and `Charge-At` attacks but only the `Goblin` can perform `Long-Range` attacks. Besides, both entities are able to take objects, but the `Ogre` is able to take heavier things due to its stronger strength. In a similar way, both units can take cover but an `Ogre` needs higher cover to hide itself because of its size.

Two special fields, in every entity, should be added to the typical *blueprints* file explained in Chapter 3. They would be the *ontType* and the *parentOnt*, where the first one would set the corresponding symbolic name for the *concept* that would represent this entity in the ontology and the second one would specify the branch or branches in which it should be added (which *concepts* would be the parents of the new *concept*). With these new fields, the system would be able to know where the new *concepts* should be added into the ontology in order to represent these new types of entities.

Once the entity had been added to the ontology in form of *concept*, the system would add information about its properties and the actions that it was able to carry out. This process would be done by iterating over the list of components that the entity had, asking them about which information would have to be injected to the corresponding *concept*. To simplify operator preconditions and postconditions and the *concept* descriptions, some auxiliary predicates could be defined on the ontology:

```
    canWalk = entity and hasSkill.walk
    canMove = canWalk or canRun or canSneak
    canTake = entity and hasSkill.take
...
```

As an example, the `Ogre` entity that appears in Figure 7.2 can be considered. When the system iterated over the components, its description would become:

```
Ogre = NPC and canWalk and canTake and hasStrength.strong ...
```

On the other hand, planning operators would correspond to basic behaviours that would be implemented as software components. Each software component would know which parameters would be required and the conditions that the current state of the world should hold in order to be applicable. However, this information is usually written procedurally in C++ or other programming language. We propose to use components that were able to describe themselves by means of preconditions and postconditions using the vocabulary in the domain ontology. That is, each component that represented a behaviour would be able to provide, though its programming interface, the planning action that described it.

WALK−TO(?who: alive, ?target: entity)
vars: ?r
pre: canWalk (?who), inRoom(?who, ?r), inRoom(?target, ?r), aloneInRoom(?who)
post: nextTo(?who, ?target)

SNEAK−TO(?who:alive, ?target: entity)
vars: ?r, ?e
pre: inRoom(?who,?r), inRoom(?e, ?r), enemyOf(?e,?who), undetected(?who)
post: canSneak(?who), nextTo(?who, ?target), undetected(?who)

TAKE(?who: alive, ?what: resource)
vars: ?w, ?s
pre: canTake(?who), nextTo(?who,?what), hasWeight(?what, ?w),
hasStrength(?who, ?s), enoughStrength(?s, ?w)
post: inInventary(?who, ?what)

MELEE−ATTACK(?who: alive, ?w: weapon, ?target: entity)
vars: ?t
pre: canMeleeAttack(?who), nextTo(?who, ?target), hasWeapon(?who, ?w),
meleeWeapon(?w), hasTechnology(?w, ?t), canHandleWeaponTech(?who, ?t)
post: #removeInstance(?target)

LONGRANGE−ATTACK(?who: alive, ?w: weapon, ?target: entity)
vars: ?t
pre: canLongRangeAttack(?who), nextTo(?who, ?target), hasWeapon(?who, ?w),
longRangeWeapon(?w), hasTechnology(?w, ?t), canHandleWeaponTech(?who, ?t)
post: #removeInstance(?target)

TAKE−COVER(?who: alive, ?c: cover)
vars: ?r, ?s1, ?s2
pre: canTakeCover(?who), uncovered(?who), inRoom(?who, ?r), inRoom(?c, ?r),
cover(?c), hasSize(?who, ?s1), hasSize(?c, ?s2), lessEqSize(?s1, ?s2)
post: covered(?who)

CHARGE−AT(?who: alive, ?w: weapon, ?target: entity)
vars: ?r
pre: canCharge(?who), detected(?who), inRoom(?who, ?r), inRoom(?target, ?r),
nextTo(?who, ?target), hasWeapon(?who, ?w), meleeWeapon(?w),
hasTechnology(?w, ?t), canHandleWeaponTech(?who, ?t)
post: #removeInstance(?target)

Figure 7.3: Planning operators corresponding to basic behaviours.

Figure 7.3 shows some of the planning actions that might be available in a game. For example, the `TAKE` operator would be generated by the `Take` component, and the precondition of the planning operator would depend on the strength of the entity that was performing the action. This is because the component would depend on the attribute strength defined in the *archetypes* file (Figure 7.2). When the planner tried to use this operator with an `Ogre`, or a `Goblin`, the planner would check if the entity was able to take things and if the entity had enough strength to take the resource. As the strengths of both entities had been previously asserted in the ontology, and due to the fact that the `Ogre` was stronger than the `Goblin`, it would be able to take heavier objects.

7.3 Implementation

With the purpose of reflecting the ideas exposed in Section 7.2, we propose the implementation shown in Code Block 6. The `IComponent` interface would be extended with two methods that should be implemented by every component class that inherits from this interface. The first method, `getOperators()` would return a list of strings in which each element of it represented a new operator describing an action that the component was able to execute. Meanwhile, the second method, `getAbilities()`, would return another list of strings where each element represented an ability or skill that the component give to the entity.

Similarly, the `Entity` class would be extended with three methods. The first two ones would only return the ontology type name (of the *concept* which was to be created) and the parents of this *concept*. Both would be taken from the *blueprints* file (Figure 7.2). The third method, `getAbilities()`, would iterate over its components invoking their `getAbilities()` method. Its only propose would be to merge the lists that they reported.

So, in order to generate the planning domain with these extensions, only two methods would be needed. The `createOperators()` method would collect all the operators that the components, which game entities had, generated. In the proposed implementation of Code Block 6, a list of components with one component of each kind would be supposed. So, the method would only iterate over this list merging all the lists reported, and then, the resultant list would be given to the planner. If this list of components did not exist, the same result could be achieve by collecting the operators of each entity, having in mind that some components could be repeated over the different entities. Consequently, the method would have to drop the duplicated operators.

The other method needed to generate the planning domain would be the `createAndFillConcepts()` method, which would iterate over the list of entities creating the *concepts* into the planner and adding their abilities. A `createConcept()` method in the planner would be supposed for creating new concepts. This method would need the *concept* name and a list names that represented its parents *concepts* in the ontology. Meanwhile, a `setAbilities()` method would be supposed into the *concepts*.

So, the process of generating the planning domain would run as follow: 1) Different entities would be created from the *blueprints* file and they would be filled from the *archetypes* file. 2) The operators would be added to the planner by the `createOperators()`

method. 3) The *concepts* of the planner would be created, and filled with their abilities, by the `createAndFillConcepts()` method.

```
class IComponent {
    ...
    virtual void spawn(bool full);
    virtual List[string] getOperators() {
        return NULL;
    }
    virtual List[string] getAbilities() {
        return NULL;
    }
    ...
};

class Entity {
    List[IComponent] _components;
    string _ontType;
    string _parentOnt;
    ...
    string getOntType() {
        return _ontType;
    }
    List[string] getParentOnt() {
        return _parentOnt;
    }
    List[string] getAbilities() {
        List[string] abilities;
        for each IComponent c in _components {
            abilities.merge(c.getAbilities());
        }
        return abilities;
    }
  ...
};

class PlanDomainGenerator {
    Planner _planner;
    List[Entity] _entities;
    List[IComponent] _kindsOfComponents;
    ...
    void createOperators() {
        List[string] operators;
```

```
        for each IComponent c in _kindsOfComponents {
            operators.merge(c.getOperators());
        }
        _planner.addOperators(operators);
    }
    void createAndFillConcepts() {
        for each Entity e in _entities {
            _planner.createConcept(e.getOntType(),e.getParentOnt());
            Concept c = _planner.getConcept(e.getOntType());
            c.setAbilities(e.getAbilities());
        }
    }
    void createPlanningDomain {
        //from blueprints and archetypes.
        _entities = createEntities()
        createOperators();
        createAndFillConcepts();
    }
    ...
}
```

Code Block 6: Pseudo-code of the implementation

7.4 Different Uses

The *reflective component based system* that we are proposing in this chapter can be used in different ways: During the design time and during the execution of the game.

During the design time, designers would use tools that help them in the difficult task of generating behaviour trees. In this way, a design tool based in this system could be created. It would help designers by giving different alternatives to achieve a goal, which the designer wanted to resolve, under specific circumstances of the environment that the designer proposed.

During the execution of the game, the game itself would use the system to create alternatives to achieves goals that were not achieved by any reason. In this way, the game engine would be more robust having the possibility of create new ways of achieving the goals when some branches of the behaviour tree had not been carried out, under special circumstances of the environment, in the way that the designers had proposed.

7.4.1 During the Design Time

Figure 7.4 summarizes our proposal to support the AI designer during the creation of behaviour trees. By means of a graphical interface, that could be a simplified version of

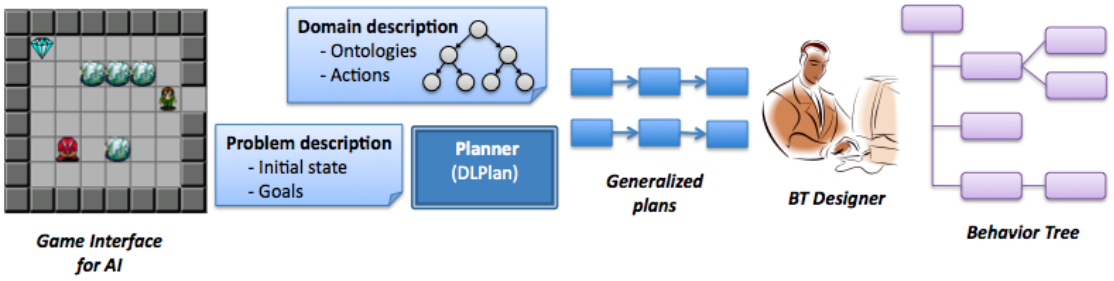

Figure 7.4: Interactive process to create Behaviour Trees

the game interface, the designer would set up a particular game scenario and some goals. Next, the system would generate the equivalent symbolic description using the planning language, and by means of a planner, it would compute all the possible plans that solved the problem. Then the designer could use these plans to complete the behaviour tree that was currently building. Let us remember that behaviour trees are useful in a broad set of scenarios and thus, this would be an interactive process in which the designer proposed different scenarios to the system and incrementally he would complete the behaviour tree using the retrieved solutions.

In a simple approach, the process, by which plans were integrated to the current behaviour tree, would be manual, i. e., the designer would be the only person responsible of changing the behaviour tree to add the new branches. In a more complicated approach, how to automate this task, or at least how to provide more support to the designer, should be considered. However, it is important to remark that the planner works with a limited model of the game, while the designer can take into account much more factors (like story plot or special situations) in order to select behaviours, modify preconditions under certain circumstances and set the priority of each alternative. The planner output, therefore, would only show different solutions that the designer should modify, validate or reject. Anyway, preconditions of the planning operators and the generalized plans computed by DLPlan would be a good source of inspiration to define the guards of new nodes in the tree.

Now we are going to expose how to build behaviour trees using our approach, i.e., taking advantage of planning techniques to support designers during the process. In previous sections we have described how, from the programming point of view, world entities would be defined by means of software components that provide different abilities, and how those components would describe themselves using preconditions and postconditions. Besides, those formal descriptions was made in terms of the vocabulary defined in an ontology that serves as a joint point between programmers and designers. Using the ontology and the component descriptions a planning domain had been described, and this way, designers was able to propose different problems to the planner with the purpose of getting all the possible solutions for concrete scenarios.

7.4.1.1 Example

Let us expose a concrete example in which a behaviour tree should be created to control a greedy goblin that had entered in a room to discover a shiny diamond in the opposite corner. Complex games usually provide intuitive interface for designers to describe concrete scenarios or even to define new levels from scratch (McNaughton et al. (2004)). So, the existence of one of these graphical interfaces to define different initial states and goals without having to deal with logical predicates but just setting items and units in the map and defining theirs attributes, would be assumed.

Let us start with the simplest situation, where the goblin and the diamond was in the same room and there was no enemies near. This goblin would be a warrior and thus it would be well armed with a short sword, a small knife, a short bow and a sling. The room, in turn, would contain some furniture: a table, two chairs and a bookcase. Although the designer did not know it, behind the scene this information would be automatically translated to a symbolic representation for the planer using the vocabulary in the ontology:

```
Goblin(goblin1), hasWeapon(goblin1, knife1), Knife(knife1),
hasWeapon(goblin1, sword1), ShortSword(sword1), ...,
inRoom(goblin1, room1), inRoom(table1, room1), Table(table1),
..., inRoom(diamon1, room1), Diamon(diamon1)
```

Now, when the designer defined the goal (the goblin gets the diamond), the planer would show all the possible plans that accomplishes it. In this case there would be only one possible plan: walk until the diamond location and take it:

1. `WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`

At this point, using the abstraction capabilities of DLPlan the system would be able to point out that this plan would be applicable in several more scenarios, because the plan only would require that the *goblin1* was an entity that can walk and take things and that it was alone in the room, and that the *diamon1* was a small item. The generalization process followed by DLPlan to reach this conclusion is based on the ontological domain definition and it is described in Sánchez-Ruiz et al. (2009).

Using this information, the designer would build a behaviour tree like the one shown in Figure 7.5, that represents the only plan available in this scenario. It is important to mention that plans generated using the planner are sequences of actions that correspond to the leaves of the behaviour tree. The definition of internal nodes in the tree to group basic actions and to represent different alternatives is responsibility of designers.

Next, the designer should complete this basic behaviour tree to make it useful in other scenarios as well. For example when there was an enemy in the same room that had already detected the goblin. This time the planner would compute several more possible plans:

1. `ChargeAt(goblin1,sword1,enemy1), WalkTo(goblin1,diamon1),`
   `Take(goblin1,diamon1)`

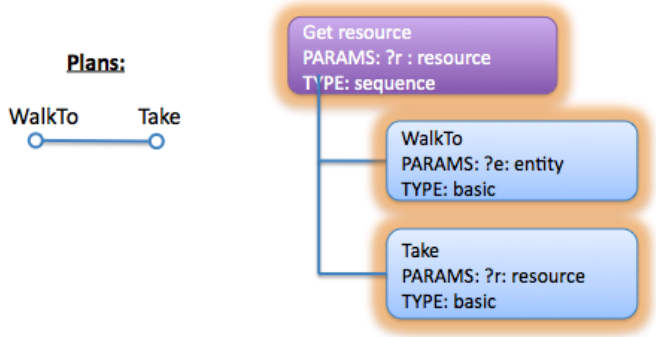

Figure 7.5: Example of a behaviour tree creation (first version)

2. `ChargeAt(goblin1,knife1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`
3. `TakeCover(goblin1,table1), LRAttack(goblin1,bow1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`
4. `TakeCover(goblin1,table1), LRAttack(goblin1,sling1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`
5. `TakeCover(goblin1,bookcase1), LRAttack(goblin1,bow1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`
6. `TakeCover(goblin1,bookcase1), LRAttack(goblin1,sling1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`

It is important to mention that during the computation of these plans the planner had performed some interesting inferences using the domain knowledge. For example, the planner had used the table and the bookcase as possible covers for the goblin because they were big enough (chairs were not), and different weapons had been classified depending on their range in melee or long range weapons.

Actually, the six generated plans would be in fact two different strategies parametrized with different values: 1) to charge against the enemy and then to take the diamond; 2) to look for a cover, to attack the enemy from the distance and then to take the diamond. Again, the system would take advantage of the abstraction capabilities of DLPlan to compute the weakest scenario in which each plan was applicable, and this way, it would reduce the six concrete plans to only two generalized plans:

1. `ChargeAt(goblin1,sword1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`
2. `TakeCover(goblin1,table1), LRAttack(goblin1,bow1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`

The first plan would be applicable if the *goblin1* was an entity that was able to charge, walk and take; the *goblin1* had a melee weapon called *sword1*; the *enemy1* was an enemy unit that had already detected the goblin; and if the *diamon1* was a small item. In the

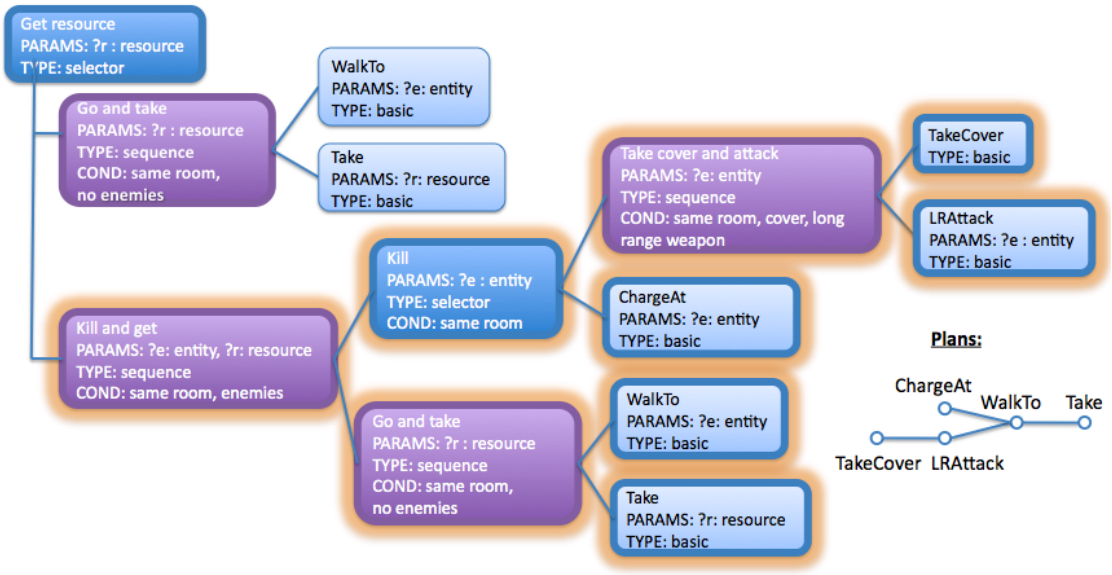

Figure 7.6: Example of a behaviour tree creation (second version)

same way, the second plan would be applicable if the *goblin1* was an entity that was able to perform the corresponding actions, it had a long range weapon *bow1*, and the *table1* was a cover of medium size.

The computation of plans and the later generalization would be performed behind the scene, and so, the designer would only see the generalized plans. Then, he would have to complete the previous behaviour tree to incorporate the new possibilities. The resulting behaviour tree could be similar to the one shown in Figure 7.6 where the new branches are highlighted. Basically, the previous tree is now a sequence node that is only applicable if there are no enemies in the room, and in order case the entity would have to kill the enemies first. Furthermore, there are two ways to kill an enemy, either charge at him or take a cover and attack him from the distance.

Finally, the designer could want to complete the behaviour tree with new branches that would be executed when there was an enemy in the room but it had not detected the goblin yet. This time, the planner would compute the following solutions:

1. `TakeCover(goblin1,table1), LRAttack(goblin1,bow1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`
2. `TakeCover(goblin1,table1), LRAttack(goblin1,sling1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`
3. `TakeCover(goblin1,bookcase1), LRAttack(goblin1,bow1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`
4. `TakeCover(goblin1,bookcase1), LRAttack(goblin1,sling1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`

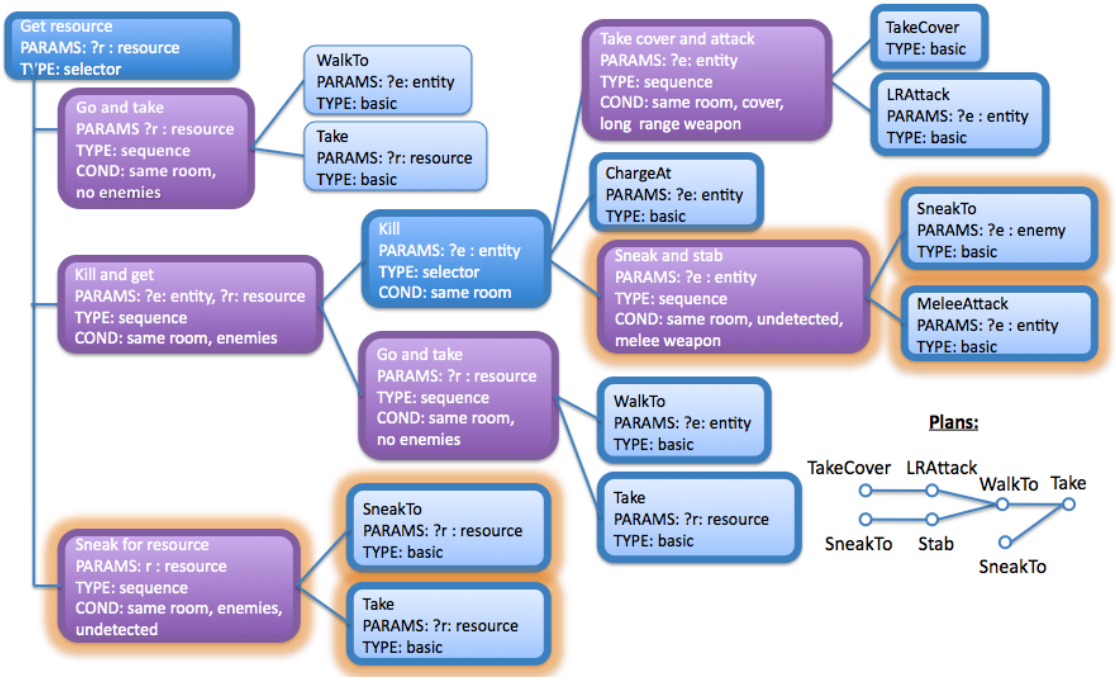

Figure 7.7: Example of a behaviour tree creation (final version)

5. `SneakTo(goblin1,enemy1), MeleeAttack(goblin1,sword1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`
6. `SneakTo(goblin1,enemy1), MeleeAttack(goblin1,knife1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`
7. `SneakTo(goblin1,diamon1), Take(goblin1,diamon1)`

In turn, these plans would be generalized in three strategies: 1) take a cover, attack from the distance, go to the diamond and take it; 2) sneak until the enemy, stab him, go until the diamond and take it; and 3) sneak until the diamond and take it (without killing the enemy).

Next, the designer would add these new alternatives to the behaviour tree, obtaining something similar to Figure 7.7. In this case, the strategy of looking for a cover and attacking using a long range weapon was already in the previous behaviour tree so the designer would only need to add the other two plans. Basically, the designer would add a new way to kill the enemy (sneak and stab) and another new way to get the resource without killing the enemy (sneak for resource).

7.4.2 During the Execution of The Game

During the execution of a game, so many unexpected things can happen, and not all of them can be anticipated by the designers. In spite of this, the final result must control

every different situation and should contemplate that the designers may forget to handle some specific situations. As it has been widely explained in Chapter 6, the behaviour trees can fail in the associations with different entities or the behaviour trees can fail because of the state of the game environment. So, to fix the problems that can happen in them, the *reflective component based system* exposed in this chapter can be adapted to deal with the failures produced during the execution of the game.

As during the design time, during the execution of the game, the system could offer different ways to achieve concrete goals under different situations of the environment. The different alternatives offered by the system would be used to fix behaviour trees that for any reason were not be executed. There would be, however, some differences in the way that the system was used. The first difference, with the system proposed for the tool specified in the previous section, would be that the system would have to produce behaviour trees automatically in order to be integrated in the game engine. This is because the game engine would need the fixes in form of behaviour trees to use them as the same way as behaviour trees designed by designers were used. So, plans given as in the previous section would not suffice.

Another difference would be that there would not be any person to choose which choices that the planer gave would have to be taken. So, in this approach, the game engine would have to choose between the different choices that the system provided. At this point, a measure of similarity would be needed to compare the different choices, offered by the planer, with the behaviour that failed during the game and in this way choosing the more similar choice to replace it.

Finally, another system to turn into logical predicates the actual state of the environment, and the parameter that the entities had, would be assumed.

Thus, to summarize, the system would work in a similar way as in the previous tool, but with the differences that the translation of the plans into behaviour trees would have to be automatic, the choice to fix the tree would be choosing automatically having into account a measure of similarity, and the environment given to the planner would not be proposed by anybody, it would be directly taken from the game engine.

Chapter 8

Conclusions

In the beginning of this thesis has been shown that authoring the AI for non-player characters (NPCs) in modern video games is an increasingly complex task. There exists a tension between designers and programmers because designers want to include their narrative story lines in the game, and it requires a big effort make by programmers because implement AI specified by good story tellers, who are not programmers, usually is a difficult task. Furthermore, the process requires some revisions because programmers do not usually get what designers really want in the first try and even though they get it, designer may want to refine what they have previously described.

Nowadays, the industry is making a big effort to separate the collaboration between both groups as much as possible, so programmers need to provide designers with mechanisms which will allow them to design history plots through character behaviours. Because of this problem, we have proposed the use of *Behaviour Tree*s in order to let designer develop whatever they want. *Behaviour Tree*s have shown in recent times that they may be one of the choices for the future to create character behaviour, but they are still being a difficult way to create behaviours for designers without programming skills.

At the same time, we have explain that creating design tools, or systems, with the purpose of developing character behaviours, is usually a tedious task because all the information that entities of the game have, must be duplicated in the field of the tool, or system. Even, the maintenance of these tools, or systems, during the developing time of the game, can turn into a nightmare when programmers are continuously adding or modifying game entities, because every change in the game should be replicated in every design tool in order to maintain them actualized.

We have chosen a *component-based system* for several reasons. It solves the principal problems presented in *object-management system*s, it is the choice of actual videogames for creating game entities and it is easy to link with *Behaviour Tree*s. The last assertion is because every component can be seen as an ability/skill that an entity has while, in a *Behaviour Tree*, *leaf*s or final *action*s are simple behaviours that the entity has to execute. So, it can be supposed that every component would be able to carry out one or more *action*s and, at the same time, every *action* would be able to be carried out by one or more components.

In that sense, and having in mind everything told in this paper, we have extended the `IComponent` interface with the purpose of avoiding replicating the information of the game entities into the developed systems and tools. In this way, the whole knowledge is kept in the same place, the components, and it is not duplicated. Consequently, the components are those responsible of providing different tools and systems with the information that they need to work, either before they have started their processing, by generating the knowledge that they need; or during their processing, by giving them answers to the queries that these tools/systems send.

We have proposed two different systems based on the concept of *Reflective Component*s and both of them provide promising results. The first one, proposed for checking associations between behaviours and concrete entities, is being developed on a serious game called JAVY 2 (Gómez-Martín, 2008) that is under development by our department (). In this approach, *Reflective Component*s can check associations both when the *Behaviour Tree*s are being developed during design time, and when the *Behaviour Tree*s are being executed. The difference between them is that at design time, entities and components are not full initialized, because different engines such as graphic or physic engines are not instantiated in order to make the system lighter. Some examples of the code is attached in Appendix A. As a future work we will continue developing this system to fully integrate it with every entity and component of the game both at design time and during the execution of the game. We could also think in developing a complex graphical tool that let designers create and check behaviour trees in an easier way.

The second proposed system is only a proposal yet, however it could be in the future a good field to research. In this way, as a future work we could consider to give a more concrete specification for this proposal with the purpose of developing them and testing which kinds of results we would achieve. Then, in order to prove the proposal, we could offer to different designers both choices: to create *Behaviour Tree*s with or without the help of the system. After that, we could measure how long it take to develop complex behaviours with or without the system and we could put face to face behaviours of different choices and measure with kind of behaviours obtains better results.

As another future work, we can think in developing character behaviours with these tools in order to create actors for interactive storytelling (Mateas and Sengers, 1999). Nowadays, environments for interactive storytelling are usually developed as a character-based multi-agent system in which every character of the story is controlled by an agent that perceive the environment, think about its possibilities re-evaluating them and acts on the environment. Consequently, the way in which different actors are created is not important and, in the same environment, characters controlled by different methods can inhabit. This way could be really useful to prove how our character reacts into an environment in which other kinds of character inhabit.

There are many ways of creating behaviours for characters in interactive storytelling. For example we can found agents directed by goals that usually are BDI agents that do not reconsider very often their goals. They can be implemented by Hierarchical Task Networks (HTNs) (Cavazza et al., 2002a,b, 2001), by languages like ABL (O.Riedl and Stern, 2006) or by similar ways. Feelings can be added to make them more realistic (Pizzi

et al., 2007; Peinado et al., 2008) and, as well, agents with multi-layer choices (Imbert and de Antonio, 2005; Spinola et al., 2008) or extended BDI model (Grimaldo et al., 2008) can be found in storytelling.

So, this would be a complete scenario to prove our proposals in the future. In theory, *Behaviour Tree*s would react well in these kinds of environments because they are directed by goals but at the same time they have the possibility of re-evaluate these goals due to many factors. Thus they would be realistic behaviour if they are well built. An incentive of proving our proposals in this kind of environment is that there exist many models to take as references and to incite us to improve our approaches, when they presented disadvantages facing with other approaches such as those that we have commented before.

To finalize let us say that the *Reflective Component*s appear a good field in which people can investigate in order to facilitate the creation of tools and system related to *component-based system*s. We focused all the paper in how *Reflective Component*s can help people in the task of creating character behaviour, but it can be extended to many different fields always that they use a *component-based system*.

Appendix A

JAVY 2 code

In this appendix the most relevant parts of the JAVY 2 code related with *Reflective Component*s can be seen. JAVY 2 (Gómez-Martín, 2008) is a serious game that is under development by our department. On this game we have developed the concrete proposal told in Chapter 6 with the purpose of knowing failures produced during associations between concrete character and *Behaviour Tree*s.

So we have selected the pieces of code that we consider more relevant. However, Classes as `Entity` class have been omitted due to their simplicity.

A.1 Messages

Firstly we present an example of a message and the base class of every message. The base class has some macros declared in order to eases the creation and the use of them. Their implementation will be omitted except of macros defined in the `CType` class. This class is used to give the same *id* to every instance of the same class and, in this way, we can know which is the class of every message.

```
namespace OIM
{
        class CType
        {
        public:
                /**
                Constructor
                */
                CType(){}

                /**
                Equality operator.
                */
```

```
                bool operator==(const CType& id) const {
                        return this == &id;
                }
        };


/**
    Type Id macros.
*/
#define DECL_TYPE \
public:\
  static OIM::CType TypeId; \
        static const OIM::CType &Type; \
  virtual const OIM::CType& getType() const; \
        virtual bool IsA(const OIM::CType &) const; \
private:

#define IMPL_TYPE(Tipo) \
  OIM::CType Tipo::TypeId; \
        const OIM::CType &Tipo::Type = Tipo::TypeId; \
  const OIM::CType& Tipo::getType() const { return Tipo::TypeId; } \
        bool Tipo::IsA(const OIM::CType &id) const
                { return id==Tipo::TypeId; }

#define IMPL_SUBCLASS_TYPE(Tipo, Superclass) \
  OIM::CType Tipo::TypeId; \
        const OIM::CType &Tipo::Type = Tipo::TypeId; \
  const OIM::CType& Tipo::getType() const { return Tipo::TypeId; } \
        bool Tipo::IsA(const OIM::CType &id) const
                { return (id==Tipo::TypeId) || Superclass::IsA(id); }

#define IS_STRICT_INSTANCE_OF(obj, Tipo) \
        ((obj).getType()==Tipo::TypeId)

#define IS_INSTANCE_OF(obj, Tipo) \
        ((obj).IsA(Tipo::TypeId))
}//OIM
```

Code Block 7: Code of the CType.h file

```
namespace OIM
{
```

```
        class CMessage : public Base::CContDeRef
        {
                DECL_TYPE();
        public:

                MSG_DECLARA_PTR(CMessage);

                /**
                Constructor
                */
                CMessage();
                /**
                Virtual Destructor. It must be implemented in children.
                */
                virtual ~CMessage() {}
        };

} // namespace OIM
```

Code Block 8: Code of the CMessage.h file

As it can be seen in the `CGoToMsg` class, the way in which messages are declared is very easy using macros. Every `MSG_DEFINE_PROPERTY` macro creates a variable and its getter and setter methods.

```
namespace OIM
{
        class CGoToMsg : public CMessage
        {
                DECL_TYPE();

        public:
                MSG_DECLARA_PTR(CGoToMsg);

                /**
                Constructor.
                */
                CGoToMsg() : _Distance(0), _Finished(false),
                        _Successfully(true) {}

                /**
```

```
                Destructor.
                */
                ~CGoToMsg() {}

                MSG_BEGIN_PROPERTIES(CGoToMsg);

                        /// Position to achieve
                        MSG_DEFINE_PROPERTY(TVector3, Destination);

                        /// Minimum distance to the object for stopping.
                        MSG_DEFINE_PROPERTY(float, Distance);

                        /// True when the action is finished.
                        MSG_DEFINE_PROPERTY(bool, Finished);

                        /// True if the goal was achieved.
                        MSG_DEFINE_PROPERTY(bool, Successfully);

                MSG_END_PROPERTIES();
        };
} // namespace OIM
```

Code Block 9: Code of the CGoToMsg.h file

```
#include "GoToMsg.h"

namespace OIM {
        IMPL_TYPE(CGoToMsg);
} // namespace OIM
```

Code Block 10: Code of the CGoToMsg.cpp file

A.2 Component

The `IComponent` interface have been extended with the `spawnInDesignTime` method in order to initialize the entity when we are at design time, and with the `canCarryOut` method in order to know which messages the component is able to carry out.

```
namespace OIM
{
        class IComponent : public CCommunicationPort {
                DECL_TYPE();
        public:
                /**
                spawn method of the component at design time
                */
                virtual bool spawnInDesignTime
                        (COIMObject* obj, OIM::CMap *map,
                        const Maps::CMapEntity *entity);

                /**
                Virtual method that every component must overwrite in
                order to specify which messages they are able to carry
                out.
                */
                virtual bool canCarryOut(CMessage* message) {
                        return false;
                }
} // namespace OIM
```

Code Block 11: Code of the IComponent.h file

Now, some examples of how these methods are implemented by specific components are shown.

```
namespace OIM
{
        bool CAvatarComponent::spawnInDesignTime
                (COIMObject* obj, OIM::CMap *map,
                const Maps::CMapEntity *entity) {
                if( !IComponent::spawnInDesignTime(obj,map,entity) )
                        return false;
                return true;
        }

        bool CAvatarComponent::canCarryOut(CMessage* message) {
                if ((IS_INSTANCE_OF(*message, OIM::CPhysicMovementMsg))
                  || (IS_INSTANCE_OF(*message, OIM::CHumanMovementMsg)))
                {
```

```
                        CMoveEntityToMsg::Ptr
                                message(MsgBuilder<CMoveEntityToMsg>());
                        return _obj->canCarryOut(message,this);
                }
                return false;
        }
} // namespace OIM
```

Code Block 12: Code of the CAvatarComponent.cpp file

```
namespace OIM
{
        bool CGraphicComponent::spawnInDesignTime
                (COIMObject* obj, OIM::CMap *map,
                const Maps::CMapEntity *entity) {
                if( !IComponent::spawnInDesignTime(obj,map,entity) )
                        return false;

                // We read the file name
                std::string modelname;
                if (entity->existsKey("modelfilename"))
                        modelname = entity->getAtribute("modelfilename");

                if (modelname.empty())
                        assert(entity->getModelo() &&
                        "Graphic Component without model!");

                return true;
        }

        bool CGraphicComponent::canCarryOut(CMessage* message) {
                return IS_INSTANCE_OF(*message, OIM::CMoveEntityToMsg) ||
                        IS_INSTANCE_OF(*message, OIM::CActivateMsg);
        }
} // namespace OIM
```

Code Block 13: Code of the CGraphicComponent.cpp file

```
namespace OIM
{
        bool CGoToComponent::spawnInDesignTime
                (COIMObject* obj, OIM::CMap *map,
                const Maps::CMapEntity *entity) {
                if( !IComponent::spawnInDesignTime(obj,map,entity) )
                        return false;
                return true;
        }
        bool CGoToComponent::canCarryOut(CMessage* message) {
                if ( IS_INSTANCE_OF(*message, OIM::CGoToMsg) &&
                        !((OIM::CGoToMsg*)message)->getFinished() ) {
                        // We have to check if there is another component
                        // in the entity that can carry out a
                        // CSteeringMsg message.
                        CSteeringMsg::Ptr msg(MsgBuilder<CSteeringMsg>().
                                Destination(
                                ((CGoToMsg*)message)->getDestination())
                                );
                        return _obj->canCarryOut(msg, this);
                }
                return false;
        }
} // namespace OIM
```

Code Block 14: Code of the CGoToComponent.cpp file

```
namespace OIM
{
        bool CLogicDoorComponent::spawnInDesignTime
                (COIMObject* obj, OIM::CMap *map,
                const Maps::CMapEntity *entity) {
                if( !IComponent::spawnInDesignTime(obj,map,entity) )
                        return false;

                if (!entity->existsKey(DURATION) &&
                        !entity->existsKey(SPEED))
                        fprintf(stderr,
                        "Warning: Door without establish speed.\n");
```

```
                if (!entity->existsKey(DISTANCE))
                        fprintf(stderr,
                        "Warning: Door without establish distance.\n");

                if (!entity->existsKey(AXIS))
                        fprintf(stderr,
                        "Warning: Door without establish axis.\n");

                return true;
        }
        bool CLogicDoorComponent::canCarryOut(CMessage* message) {
                if( IS_INSTANCE_OF(*message, OIM::CTriggerMsg) ||
                        IS_INSTANCE_OF(*message, OIM::CUseMsg) ) {
                        // We need to move both physic entity and graphic
                        // entity. At least one of them because if both
                        // are not, a door does not make sense.
                        CMoveEntityToMsg::Ptr message(
                                MsgBuilder<CMoveEntityToMsg>()
                                );
                        return _obj->canCarryOut(message,this);
                }
                if( IS_INSTANCE_OF(*message, OIM::CMouseMsg) ) {
                        // It is only accepted if there are an action
                        // to do.
                        return ((CMouseMsg*)message)->getLeftClicked() ||
                              ((CMouseMsg*)message)->getRightClicked() ||
                              ((CMouseMsg*)message)->getTouched();
                }
                if( IS_INSTANCE_OF(*message, OIM::CLockedMsg) ) {
                        return true;
                }
                return false;
        }
} // namespace OIM
```

Code Block 15: Code of the CLogicDoorComponent.cpp file

```
namespace OIM
{
        bool CSoundComponent::spawnInDesignTime
                (COIMObject* obj, OIM::CMap *map,
```

```
                const Maps::CMapEntity *entity) {
                if( !IComponent::spawnInDesignTime(obj,map,entity) )
                        return false;
                const char* pcFile;

                if (entity->existsKey("file"))
                        pcFile = entity->getAtribute("file");
                else
                        pcFile = "";

                assert(strlen(pcFile) < sizeof(_filename));

                strcpy(_filename, pcFile);

                return true;
        }

        bool CSoundComponent::canCarryOut(CMessage* message) {
                if (IS_INSTANCE_OF(*message, OIM::CPlayMsg))
                {
                        // We have to look if the file can be played
                        // by the Sound Server
                        std::string file =
                                ((CPlayMsg*)message)->getFile();
                        if( archivo.empty() )
                                return Sound::CServer::GetPtrSingleton()
                                        ->HasEfect(_filename);
                        else
                                return Sound::CServer::GetPtrSingleton()
                                        ->HasEfect(file);
                }
                return false;
        }
} // namespace OIM
```

Code Block 16: Code of the CSoundComponent.cpp file

Bibliography

Atkin, M. S., King, G. W., Westbrook, D. L., Heeringa, B. and Cohen, P. R. Hierarchical agent control: a framework for defining agent behavior. *Fifth international conference on Autonomous agents, Montreal, Canada*, 2001.

Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D. and Patel-Schneider, P. F., editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, New York, NY, USA, 2003. ISBN 0-521-78176-0.

Borovikov, I. and Kadukin, A. *AI Game Programming Wisdom 4*, chapter Building a Behavior Editor for Abstract State Machines. Steve Rabin, 2008. ISBN 1-584-50523-0.

Buchanan, W. *Game Programming Gems 5*, chapter A Generic Component Library. Charles River Media, 2005. ISBN 1-584-50352-1.

Cavazza, M., Charles, F. and J.Mead, S. Agents' interaction in virtual storytelling. *Intelligent Virtual Agents (IVA), Madrid, Spain*, 2001.

Cavazza, M., Charles, F. and J.Mead, S. Character-based interactive storytelling. *IEEE Intelligent Systems, Volume 17*, 2002a.

Cavazza, M., Charles, F. and J.Mead, S. Emergent situations in interactive storytelling. *Symposium on Applied Computing (SAC), Madrid, Spain*, 2002b.

Cutumisu, M., McNaughton, M., Szafron, B., Roy, T., Onuczko, C., Schaeffer, J. and Carbonaro, M. A demonstration of the scriptease approach to ambient and perceptive npc behaviors in computer role-playing games. *Intelligent Technologies for Interactive Entertainment, First International Conference (INTETAIN), Madonna di Campiglio, Italy*, 2005.

Cutumisu, M., Szafron, D., Waugh, J. S. D., Onuczko, C., Siegel, J. and Schumacher, A. Scriptease - motivational behaviors for interactive characters in computer role-playing games. *Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference (AAAI), Boston, Massachusetts, USA*, 2006.

DILLER, D. E., FERGUSON, W., M.LEUNG, A., BENYO, B. and FOLEY, D. Behavior modeling in commercial games. *Behavior Representation in Modeling and Simulation (BRIMS)*, 2004.

DYBSAND, E. *AI Game Programming Wisdom*, chapter A Finite-State Machine Class. Charles River Media, 2002. ISBN 1-584-50077-8.

FU, D., HOULETTE, R., JENSEN, R. and BASCARA, O. A visual, object-oriented approach to simulation behavior authoring. *Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC)*, 2003.

GARCÉS, S. *AI Game Programming Wisdom III*, chapter Flexible Object-Composition Architecture. Charles River Media, 2006. ISBN 1-584-50457-9.

GOMAN, B., THURAU, C., BAUCKHAGE, C. and HUMPHRYS, M. Bayesian imitation of human behavior in interactive computer games. *International Conference on Pattern Recognition (Christian Thurau), Hong Kong, China*, 2006.

GÓMEZ-MARTÍN, M. A. *Arquitectura y metodología para el desarrollo de sistemas educativos basados en videojuegos*. Phd, Universidad Complutense de Madrid, 2008.

GRIMALDO, F., LOZANO, M., BARBER, F. and VIGUERAS, G. Simulating socially intelligent agents in semantic virtual. *The Knowledge Engineering Review, Volume 23*, 2008.

HAREL, D. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 1987.

HOULETTE, R. and FU, D. *AI Game Programming Wisdom II*, chapter The Ultimate Guide to FSMs in Games. Charles River Media, 2004. ISBN 1-584-50289-4.

IMBERT, R. and DE ANTONIO, A. An emotional architecture for virtual characters. *International Conference on Virtual Storytelling, ICVS 2005, Strasbourg, France*, 2005.

ISLA, D. Handling complexity in the Halo 2 ai. In *Game Developers Conference*. 2005.

ISLA, D. Halo 3 - building a better battle. In *Game Developers Conference*. 2008.

KELLY, J.-P., BOTEA, A. and KOENIG, S. Offline planning with hierarchical task networks in video games. *Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE), Stanford, California, USA*, 2008.

LAKOS, J. *Large Scale C++ Software Design*. Addison Wesley, 1996. ISBN 0-201-63362-0.

LLANSÓ, D., GÓMEZ-MARTÍN, M. A. and GONZÁLEZ-CALERO, P. A. Self-validated behaviour trees through reflective components. *Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE), Stanford, California, USA, in press*, 2009.

MATEAS, M. and SENGERS, P. Introduction to the narrative intelligence symposium. *AAAI Fall Symposium on Narrative Intelligence*, 1999.

MCNAUGHTON, M., CUTUMISU, M., SZAFRON, D., SCHAEFFER, J., REDFORD, J. and PARKER, D. Scriptease: Generating scripting code for computer role-playing games. In *ASE*, pages 386–387. 2004.

MUÑÓZ ÁVILA, H., H. Y HOANG. *AI Game Programming Wisdom III*, chapter Coordinating Teams of Bots with Hierarchical Task Network Planning. Charles River Media, 2006. ISBN 1-584-50457-9.

NAKANO, A., TANAKA, A. and HOSHINO, J. Imitating the behavior of human players in action games. *Entertainment Computing (ICEC), Cambridge, UK*, 2006.

ONTAÑÓN, S., MISHRA, K., SUGANDH, N. and RAM, A. Case-based planning and execution for real-time strategy games. *ICCBR Belfast, Northern Ireland, UK*, 2007.

O.RIEDL, M. and STERN, A. Believable agents and intelligent story adaptation for interactive storytelling. *3rd International Conference on Technologies for Interactive Digital Storytelling and Entertainment, Darmstadt, DE.*, 2006.

PEINADO, F., CAVAZZA, M. and PIZZI, D. Revisiting character-based affective storytelling under a narrative bdi framework. *International Conference on Interactive Digital Storytelling (ICIDS), Erfurt, Germany*, 2008.

PIZZI, D., CAVAZZA, M. and LUGRIN, J.-L. Extending character-based storytelling with awareness and feelings. *International Conference on Autonomous Agents, Honolulu, Hawaii*, 2007.

PRIESTERJAHN, S. Imitation- based evolution of artificial game players. *Genetic and Evolutionary Computation Conference, GECCO, Atlanta, GA, USA*, 2008.

PRIESTERJAHN, S., KRAMER, O., WEIMER, A. and GOEBELS, A. Evolution of human-competitive agents in modern computer games. *IEEE Congress on Evolutionary Computation (CEC'06)*, 2006.

PRIESTERJAHN, S., WEIMER, A. and EBERLING, M. Real-time imitation-based adaptation of gaming behaviour in modern computer games. *Genetic and Evolutionary Computation Conference, GECCO, Atlanta, GA, USA*, 2008.

RENE, B. *Game Programming Gems 5*, chapter Component Based Object Management. Charles River Media, 2005. ISBN 1-584-50352-1.

RICHARDS, R. A., HOULETTE, R. T. and MOHAMMED, J. L. Distributed satellite constellation planning and scheduling. *Florida Artificial Intelligence Research Society Conference (FLAIRS), Key West, Florida, USA*, 2001.

ROSADO, G. *AI Game Programming Wisdom II*, chapter Implementing a Data-Driven Finite-State Machine. Charles River Media, 2004. ISBN 1-584-50289-4.

SÁNCHEZ-RUIZ, A. A., GONZÁLEZ-CALERO, P. A. and DÍAZ-AGUDO, B. Abstraction in Knowledge-Rich Models for Case-Based Planning. In *Proc. of International Conference on Case-Based Reasoning*. 2009.

SÁNCHEZ-RUIZ, A. A., LLANSÓ, D., GÓMEZ-MARTÍN, M. A. and GONZÁLEZ-CALERO, P. A. Authoring behaviour for characters in games reusing abstracted plan traces. *International Conference on Intelligent Virtual Agents (IVA), Amsterdam, Holland, in press*, 2009a.

SÁNCHEZ-RUIZ, A. A., LLANSÓ, D., GÓMEZ-MARTÍN, M. A. and GONZÁLEZ-CALERO, P. A. Authoring behaviours for game characters reusing automatically generated abstract cases. *Workshop on Case-Based Reasoning for Computer Games(CBRCG), Seattle, Washington, USA, in press*, 2009b.

SPINOLA, J., IMBERT, R., MEDINILLA, N., DE ANTONIO, A. and GUDWIN, R. Una capa social para cognitiva: interacción cooperativa entre agentes en entornos virtuales. *II Jornadas sobre Realidad Virtual y Entornos Vituales - JOREVIR, Albacete, España*, 2008.

SZILAS, M. Becool: Toward an author friendly behaviour engine. *4th International Conference on Virtual Storytelling (ICVS), Saint-Malo, France*, 2007.

VALVE SOFTWARE. Half life. 1998.

VIRMANI, S., KANETKAR, Y., MEHTA, M., ONTAÑÓN, S. and RAM, A. An intelligent ide for behavior authoring in real-time strategy games. *Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE), Stanford, California, USA*, 2008.

W3C CONSORTIUM. OWL web ontology language guide. W3C recommendation. http://www.w3.org/tr/owl-guide/. 2004.

WEST, M. Evolve your hiearchy. *Game Developer*, vol. 13(3), pages 51–54, 2006.

Autorización

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Reflective Components for Designing Behaviour in Video Games", realizado durante el curso académico 2008-2009 bajo la dirección de Pedro Antonio González Calero y Marco Antonio Gómez Martín en el Departamento de Ingeniería del Software e Inteligencia Artificial, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Fdo: David Llansó García.
Madrid a 22 de Junio de 2009.