SAPIENZA
UNIVERSITÀ DI ROMA

SCHOOL OF ENGINEERING IN COMPUTER SCIENCE

MASTER OF SCIENCE IN ENGINEERING IN COMPUTER SCIENCE

Master's Thesis

Non-Player Character Behavior Composition in Unity Game Engine

Candidate	**Advisor**
Stefano Cianciulli	Prof. Stavros Vassos
Candidate ID Number	**Co-Advisor**
1193325	Prof. Giuseppe De Giacomo

March 2013
Academic Year 2012/2013

Eventuale Dedica

SOMMARIO

In questa tesi di laurea magistrale viene studiato in che modo è possibile applicare la tecnica di Intelligenza Artificiale denominata composizione dei comportamenti ad uno scenario di tipo videoludico per risolvere alcuni dei problemi che sorgono durante la realizzazione di un videogioco come, ad esempio, coordinare i personaggi non giocanti per realizzare un desiderato comportamento collettivo. Viene fornito un framework realizzato per il motore di gioco Unity, che è stato usato per la visualizzazione del concetto di composizione dei comportamenti e per la conduzione dei primi esperimenti di fattibilità dell'applicazione del concetto in uno scenario di tipo videoludico.

Viene inoltre presentato un web service di tipo RESTful, realizzato utilizzando il linguaggio di programmazione Java e volto a fornire la computazione di composizione dei comportamenti as-a-service, in cui tutta l'interazione con il server viene veicolata per mezzo della spedizione e della ricezione di messaggi HTTP in accordo con i principi REST. Questo web service viene rilasciato come uno strumento di tipo cloud-based che gli sviluppatori di videogiochi possono impiegare per l'organizzazione dei personaggi non giocanti in un videogioco, o per realizzare sistemi di interactive storytelling in cui la trama si svolge rispondendo dinamicamente alle decisioni del giocatore.

ABSTRACT

In this master thesis, we investigate how the Artificial Intelligence technique called behavior composition can be employed in a video game-like scenario to address some of the challenges that arise during the development, such as coordinating the non-player characters to realize a desired collective behavior. We provide a framework developed within the Unity game engine we used to visualize the behavior composition concept and to conduct the first experiments of feasibility of the application of the concept to a video game scenario.

Furthermore, we provide a RESTful web service we realized using the Java programming language to provide the computation of behavior compositions as-a-service, in which the interaction with the server is carried out by sending and receiving HTTP messages according to the REST principles. We release this as a cloud-based tool that game developers can employ to organize the non-player characters in a video game, or to design interactive storytelling systems with a dynamic unfolding of the storyline.

CONTENTS

CHAPTER 1

INTRODUCTION

In modern video games, non-player characters play a central role in creating a captivating, believable environment in which the player immerses and interacts. One of the most common techniques used in video games for providing non-player characters with an "intelligence" is using reactive methods for behavior, commonly equivalent to *finite state machines*. Such techniques decompose and define the behavior of a non-player character in terms of states, which represent possible states of "mood" or "attitude" of the character, and transitions, which describe the actions or the conditions that make the non-player character change his current "internal" state to a different one. On a different track, artificial intelligence methods are employed to equip the storyline director of the game with decision making capabilities, aiming for a more dynamic experience. The actions of the player and non-player characters then are often taken into account in a similar fashion for decision making at the level of storyline.

In this master thesis, we will explore the opportunities that arise by applying the artificial intelligence method of behavior composition in a video game setting, in order to address some of these challenges. The behavior composition is a novel technique that is concerned with combining a collection of individual, partially controllable behaviors (that are formalized using transition systems), executing within a shared, partially predictable, but fully observable, environment, in such a way that these behaviors, collectively, act to realize a fully controllable target behavior specified by the user.

Over the last few years, with the increasing adoption of the Service-Oriented Computing (SOC) paradigm, a wide variety of algorithms, tools, and data has been exposed for public use as web services designed and implemented following the principles defined in the REST (Representational State Transfer) architectural style. The REST style is the one upon which the World Wide Web is built, and takes as the key abstraction of information the concept of *resource*; using the notion of *hyperlinks*, and sending requests by means of HTTP messages and verbs, the client of a RESTful application is able to perform the basic CRUD (Create, Read, Update, and Delete) operations of persistent storage on resources maintained across a network.

In this master thesis, we follow this emerging trend of open APIs and explore the opportunities that arise by exposing the behavior composition method as a RESTful web service. We will investigate how behavior composition-as-a-service can be used as a cloud-based tool available to game developers for organizing the interactions of the non-player characters in a video game, and the dynamic unfolding of the storyline. In particular, this thesis has the following objectives:

- Design a RESTful web service for providing behavior composition, and develop the web service based on the specification using the Java programming language;
- Develop a framework within the Unity game engine for experimenting with the web service in a real video game development setting;
- Provide examples that show novel uses of behavior composition in a video game scenario.

The rest of this master thesis is organized as follows:

- In Chapter 2, **Literature Review**, we will present the main techniques and software components currently involved in the development of a video game concerning artificial intelligence and decision making. Furthermore, we will also summarize the theoretical foundations upon which the artificial intelligence technique of behavior composition, the concept we applied to automate the process of coordination of actions between the non-player characters, is based.

- In Chapter 3, **Adopted Technology**, we will present the game engine Unity, along with the essential concepts that are required for a complete understanding of this thesis, and the technical demo Angry Bots, from which the basic assets such as levels, textures, 3D models and scripts are taken. In this chapter, we will also introduce the REST architectural style, explaining the main principles it is based upon and showing some practical applications for this architecture, also using the web services exposed by Dropbox as a concrete example;
- In Chapter 4, **The Angry Bots patrolling domain**, we will describe the basic framework we developed within the Unity game engine to experiment with the application of the behavior composition method in a video game development environment, showing and explaining in detail all the classes and scripts developed for this purpose;
- In Chapter 5, **NPC Behavior Composition using SM4LL**, we will introduce the Roman model for service modeling in Service-Oriented Computing, in which the services expose their behavioral features by means of transition systems. In this chapter, we will also report the results of a first application of the behavior composition method to a video game-like setting realized through the usage of the composition engine employed in the SM4LL Project;
- In Chapter 6, **Behavior Composition as a RESTful Web Service**, we will describe the basic architecture and the Application Programming Interface (API) of JaCO, a RESTful Web Service for behavior composition written in Java and based on the behavior composition engine provided by Alberto Iachini, which is in turn based on JTLV by Yaniv Sa'ar;
- In Chapter 7, **An example of using JaCO in Unity**, we will describe how we recast the Angry Bots patrolling example described in Chapter 4 to use the JaCO Web Service.

CHAPTER 2

LITERATURE REVIEW

2.1 Reactive behavior for NPCs in video games

The expression *non-player character* (or *NPC*) refers to any character present in a video game that is not directly controlled by the player, and hence usually controlled by the computer through artificial intelligence. One of the first successful examples of non-player characters in the history of video games is the case of *Blinky*, *Pinky*, *Inky* and *Clyde*, the four "ghosts" chasing the player in the well-known game **Pac-Man** [Midway Games West, Inc., 1979].

The behavior that non-player characters usually adopt in video games is of the *reactive* type; in the sense that they base their decision-making (e.g., start chasing the player, retreat to a safe location, etc.) solely on what they currently observe within the game environment (e.g., player's and his level of health).

In the following subsections, the main two widespread approaches to realize reactive behaviors for non-player characters will be presented: *finite state machines* and *behavior trees*. The images, as well as the main concepts between the two techniques, are taken from [Millington and Funge, 2009].

2.1.1 Finite State Machines

Finite state machines is one of the first techniques used for the creation of "intelligent" characters and, along with other additions and a technique called *scripting*, still makes up the majority of decision-making systems used in current games. For example, the ghosts in Pac-Man are realized through finite state machines, as well as the enemies that populate the dystopian City 17 in **Half-Life 2** [Valve Corporation, 2004].

A finite state machine is a structure composed of *states*: for each state we have an associated action or behavior and, as long as the character remains in the same state, it will keep on performing the same action. States are connected between themselves by *transitions*: for each transition we have an associated set of conditions; when a transition's conditions are met, the transition is said to *trigger*, and can lead to the execution of a particular action; when the transition causes the finite state machine to move from one state to another, the transition is said to *fire*. In *hard-coded* finite state machines, the descriptions of the states and the conditions for the transitions are part of the game code: the states are usually represented with *enumerated values*, and inside the update function there is a block of code for each possible state.

The traditional state machines are expressive and powerful but have some problems in presence of *alarm behaviors*, in which there is the necessity to reach a particular state regardless of the state the character was previously in, and after to return in the earlier state. With "pure" state machines, this requirement would lead to an exponential growth in the number of the states; for these situations *hierarchical* state machines, in which a single state could represent a complete state machine in its own right, are more suitable.

2.1.2 Behavior Trees

Behavior trees are a synthesis of some techniques already used in artificial intelligence, such as hierarchical state machines and decision trees, combined in a way that it easy to understand and easy to create. One of the first high-profile video games that used extensively the behavior trees was **Halo 2** [Bungie Software, 2004].

The main building block of behavior trees is represented by a *task*. A task can be something as simple as looking the value of a variable in the game state, or a more

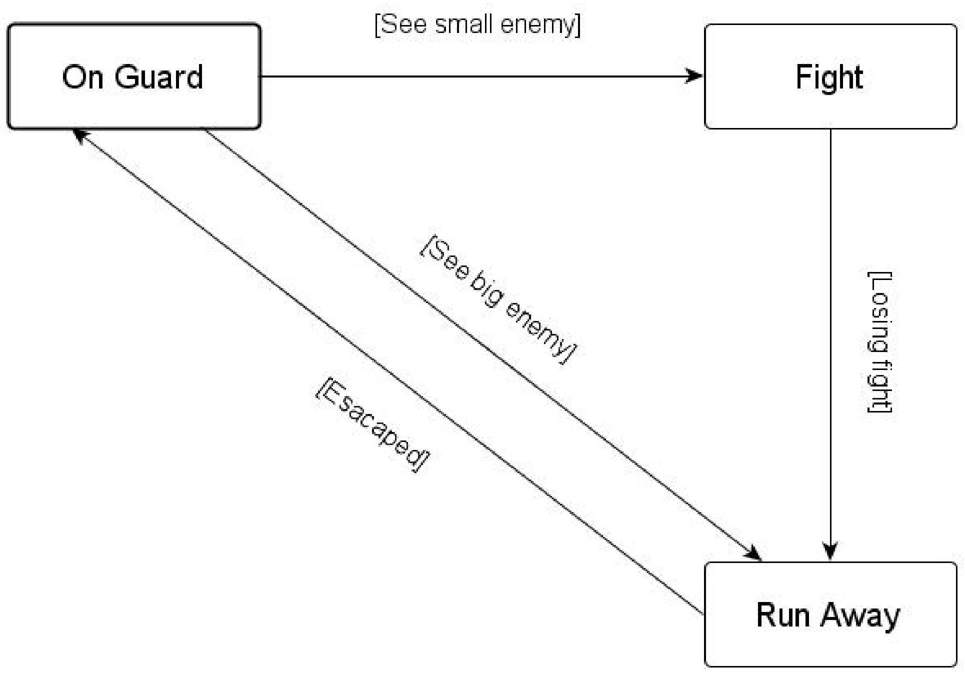


Figure 2.1: An example of Finite State Machine

complex one composed by sub-trees of other tasks. All tasks are self-contained, and have the same interface: they are given some CPU to execute, and usually return a status code indicating either the *success* or the *failure* of the execution. This common interface allows the developers to easily construct behavior trees without knowing the details of the implementation of each task.

The simplest forms of behavior trees will be made up of three basic kinds of tasks:

- Conditions: this kind of task tests some property of the game (e.g., if a certain door is open or closed), and returns *success* if the condition is met and *failure* otherwise;

- Actions: there could be Actions for animations, moving characters, changing the state of the game, and so on; usually Actions will succeed, although it is possible to have Actions that fail if they can't complete;

- Composite: Composites makes up most of the branches, because they manage a collection of children tasks (Conditions, Actions, or other Composites), and their status code depends on the status code returned by their children. There are two sub-types of Composite tasks: the *Selector*, that will return a *success* as soon as one of the children tasks runs successfully, and the *Sequence*, that will return with a *failure* status if one of the children tasks returns a *failure* status code.

More advanced kinds of tasks include the non-deterministic versions of the *Selector* and the *Sequence* tasks, used to make the behavior of the non-player character less predictable but still consistent; *Decorators*, that have exactly one child task and modify their behavior in some way (e.g., they return the opposite status code); and *Parallel*, similar to the *Sequence* task but with the significant difference that it runs its children tasks concurrently until one of them fails, causing the *failure* of the *Parallel* task as a whole.

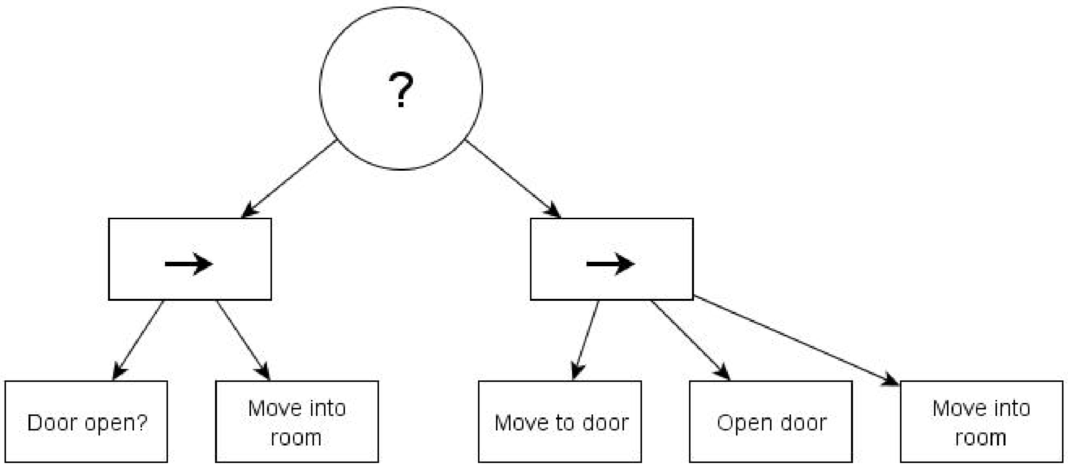


Figure 2.2: An example of Behavior Tree

More details and examples can be found in [Millington and Funge, 2009], from which also the material presented here is taken.

2.2 Proactive behavior for NPCs in video games

In the same time period that Halo 2 employed behavior trees to specify elaborate reactive behavior for NPCs, another commercial AAA title used a decision-making technique that is inspired from the artificial intelligence research technique of classical planning. The first-person shooter game **F.E.A.R.** [Orkin, 2006] employed a simplified form of STRIPS planning [Fikes and Nilsson, 1971] to guide the behavior of NPCs that is typically referred to in the game industry as Goal Oriented Action Planning (GOAP). GOAP and STRIPS essentially specify a proactive behavior in the sense that an NPC thinks before acting, deliberating about which goal is the best to execute and which sequence of actions may be performed in order to achieve it.

In the area of classical planning one is faced with the following task: given i) a complete specification of the initial state of the world, ii) a set of action schemas that describe how the world may change, and iii) a goal condition, one has to find a sequence of actions such that when applied one after the other in the initial state, they transform the state into one that satisfies the goal condition.

In STRIPS [Fikes and Nilsson, 1971], the representation of the initial state, the action schemas, and the goal condition is based on literals from predicate logic. The *initial state* is specified as a set of positive ground literals. This provides a complete specification of the state based on a closed-world assumption. That is, for all ground literals not included in the set, the negative version of the literal is assumed to hold. The *action schemas* specify the available actions in the domain as well as their preconditions and effects using sets of ground literals. In the case of preconditions a set of positive literals specify what needs to be present in the state representation in order for the action to be executable, and for the effects of an action, a set of positive and negative literals specify how the state should be transformed after action execution: all the positive literals in the set of effects are added in the set describing the state, and all negative literals are removed. A *goal condition* is also a set of positive ground literals. The intuition is that the goal is satisfied if all the literals listed in the goal condition are included in the set that describes the state. A *solution* then to a planning problem is a sequence of actions such that if they are executed starting from the initial state, checking for corresponding preconditions, and applying the effects of each action one after the other, they lead to a state satisfying the goal condition.

Even though the implementation of the planner in **F.E.A.R.** kept only a minimal aspect of the STRIPS characteristics, it demonstrated that with the hardware resources available it has become feasible to employ a deliberation method that is based on real-time search in a commercial video game. In fact, more than this, it demonstrated that such techniques may offer added value to the video game industry as the game **F.E.A.R.** received a lot of attention and awards for excellent AI performance.

Nonetheless, since the time that these two paradigms appeared in the FPS game industry, real-time planning has not followed a similar recognition as behavior trees have, for the purposes of specifying NPC behavior. Unlike finite state machines and behavior trees which rely on concepts that are easy to tweak and customize, real-time planning has proven to be a more difficult tool to use, especially for non-experts.

Another technique that is similar in spirit to classical planning is the use of Hierarchical Task Networks (HTN) [Erol, Hendler, and Nau, 1994]. In order to reach the goal, it is divided into smaller ones, easier to solve, until it is decomposed into simple actions that can be executed directly. A well-known game that uses HTN planning is the first-person shooter game **Unreal Tournament** [Hoang, Lee-Urban, and Héctor Muñoz-Avila, 2005; Héctor Muñoz-Avila and Fisher, 2004]. In Unreal Tournament, HTNs are used to coordinate the behavior of various non-player characters to execute goals collaboratively.

An interesting approach that falls in this category is the development and evaluation of an HTN planning library which used as a testbed a well known Role Playing Game, **The Elder Scrolls VI: Oblivion**. This system exported as a planning solution behavior scripts for agents, scripted in the language of this game. One main the main motives of this approach was also the increasing need to produce dynamic and automated behavior for agents according to the plan extracted given that there are numerous states in which the game environment is [Kelly, Botea, and Koenig, 2008].

Finally, some other related work that has been performed in the planning community is reported in [Bartheye and Jacopin, 2008; Michael, 2004; Ontanon, Mishra, Sugandh, and Ram, 2010].

2.3 Interactive storytelling in video games

Interactive storytelling is a form of digital entertainment in which the user influences a pre-defined storyline through actions, either by instructing the story's protagonist with commands, or acting as a general director of events in the narrative. The typical trade-off that arises while designing or developing an interactive storytelling (or *interactive narrative*) system is the one between limiting the player to choose only the actions that are consistent with the pre-determined plot and granting to the player the highest possible degree in movements and choices. Fewer limitations result in a richer interaction of the player with the game environment, but can also lead to behaviors that are harmful for the development of the plot, which may halt unexpectedly.

The two main approaches employed in the realization of interactive narrative experiences are:

- to hand-craft a structure of nodes, often in the form of a graph, where each node is a chunk of the storyline (such as a plot event) and the connections between nodes represents paths upon which the story unfolds. The player is given the ability to traverse the graph selecting two or more options (the outgoing links) at each choice-point, and the resulting sequence of visited nodes constitutes the experience of narrative;

- to create a procedural simulation, i.e. a virtual world containing elements (objects, environments, non-player characters), each with its own state and behavior. In this framework, the player is just one of the elements present in the world, and thus is subject to the evolution of the game world.

The advantage of the first approach is that it offers to the player a well-formed, efficient, and well-paced experience, while the main strength of the second idea is that it provides the player an high degree of agency and freedom. Two interesting practical examples of interactive storytelling applied to video games are represented by *Haunt 2* and *Façade*.

Haunt 2 [Magerko and Laird, 2004] is a point-and-click adventure in which an *Interactive Drama Architecture* acts like a human "dungeon-master": it has a pre-written story and tries to guide the player to unfold the plot, and at the same time attempts to predict his future behavior for steering him away from executing actions that may endanger the progression of the adventure. To forecast the player's future actions, the Interactive Drama Architecture stores a copy of the game state as it is after the occurrence of each plot point, and starting from this knowledge it tries to simulate how the world would respond to possible future actions by the player; the model could return a *success* if the player's actions satisfy the pre-conditions of an active plot points, or *failure* if no active points are fulfilled. When the system recognizes a real (or hypothesized) problem with the flow of the story, the director is able to dynamically modify the world to get the adventure back on track, for example creating objects or moving them in locations not yet explored by the player: in this way, also if he altered an important object in an irreversible manner, the director still has the possibility to recover from this difficult situation.

Façade [Mateas and Stern, 2003] is an interesting research experiment that involves not only the interactive storytelling, but also other artificial intelligence fields such as the natural language processing. In Façade you act, with your name and gender, as a long-term friend of Grace and Trip, a couple in their early thirties. During a meeting with them, you eye-witness an high-conflict argument between the two, and you are required to side with either Grace or Trip by taking an active role in the discussion, pushing it one way or the other. With respect to the dichotomy between hand-crafted stories and world simulations previously discussed, Façade tries to pursue a hybrid approach, maximizing the strengths of each technique while minimizing the drawbacks. The Façade architecture organizes the content of the dramatic story in a hierarchical structure: the main components in which the plot is decomposed are represented by *beats*, which are small segments that typically involve actions and reactions for the two characters; each beat contains some *beat goals* that, once achieved, make the plot unfold and present a new situation to the player; each beat goal contains behaviors that the framework dynamically organizes into an *active behavior tree*.

2.4 Game engines for development

The *game engine* is one of the most important software components needed during the development of a video game. The core functionalities often present in a game engine include:

- a *rendering engine*, used to draw the 2D or 3D graphics on the screen;
- a *physics engine*, whose responsibility is to give a realistic simulation of a physics system, with focus especially on gravity and collision detection among the objects in the game world;
- a *scene graph*, that is a data structure that stores and manages the position of the game objects into the scene;
- other components for *animations*, *artificial intelligence*, *networking*, etc.

The game engines are sometimes called *middleware* because of their ability to provide platform abstraction, allowing the same game to run on different platforms (PCs, video game consoles, handheld devices, etc.) with few, if any, changes to the source code. Often, game engines are also designed with a component-based architecture, that allows specific subsystems of the engine to be extended or replaced with more specialized components.

The first examples of general-purpose game engine are represented by some 2D game creation systems developed in the 1980s, such as the **Pinball Construction Set** [1] [Electronic Arts, 1983], **Garry Kitchen's GameMaker** [2] [1985], and **RPG Maker** [3] [ASCII, 1988]. Before the launch of these products, games were typically developed as a single entity, heavily relying on the *kernel* display routine. The game engines as we know them today, with a clear distinction between a reusable software component (the game engine) and the set of characters, levels, and weapons (the game *content*, or game *assets*) related to a specific game, arose in the mid-1990s, especially driven by the expansion of

[1] http://en.wikipedia.org/wiki/Pinball_Construction_Set
[2] http://en.wikipedia.org/wiki/Garry_Kitchen's_GameMaker
[3] http://en.wikipedia.org/wiki/Rpg_Maker

a genre of video games called *first-person shooters* (or *FPS*). The most important examples of engines from that period are the **idTech 1** upon which **Doom** [id Software, 1993] was constructed, the **Build** engine developed by Ken Silverman for the video game **Duke Nukem 3D** [3D Realms, 1995], and the **Quake engine**, written to power the game **Quake** [id Software, 1996].

Nowadays, a large number of game engines are available, both proprietary (i.e., developed for the usage in a particular video game and then released under a specific license) and freeware. A noticeable trend in the gaming industry is also to design and develop video games to be *moddable*, allowing the users to modify the assets present in the video game or to create some new, giving them the opportunity to create entirely different games from the original one. A few of the most widespread game engines currently in use today are:

- CryEngine 3[4], by Crytek;
- Rockstar Advanced Game Engine (RAGE), by Rockstar Games;
- Shiva3D[5], by Stonetrip;
- Source[6], by Valve Software;
- Torque3D[7], by GarageGames;
- Unity[8], by Unity Technologies;
- UnrealEngine 3[9], by Epic Games.

In this thesis we will be using the Unity game engine as a platform to apply and illustrate an artificial intelligence technique for decision making that has not been investigated in a video game setting.

[4]`http://www.mycryengine.com/`
[5]`http://www.stonetrip.com/`
[6]`http://source.valvesoftware.com/`
[7]`http://www.garagegames.com/products/torque-3d`
[8]`http://unity3d.com/`
[9]`http://www.unrealengine.com/`

2.5 The Behavior Composition Problem

In this section, we present the required background related to the problem of behavior composition in artificial intelligence. This will provide the theoretical foundation for the service we develop, to be used in decision making in video games.

The behavior composition problem is concerned with the realization of a fully controllable target behavior, starting from a collection of individual, partially controllable behaviors executing within a shared, partially predictable, but fully observable, environment. The available behaviors represent existing accessible devices or components, while the target behavior stands for a desired, but virtual (i.e., non-existing) component. The available and the target behavior, as well as the environment, are formalized using finite state transitions systems: they are composed by states and transitions, and actions cause the transition system to move from one state to another. The solution of the behaviour composition problem is represented by a so-called controller, that will effectively command the available behaviors in such a way that, collectively, they act like the target behaviour.

In the next sections we present the behavior composition framework as it is studied in [De Giacomo, Patrizi, and Sardiña, 2007; De Giacomo and Sardiña, 2007; Sardiña, Patrizi, and De Giacomo, 2008].

2.5.1 Components: Environment, Behaviors, Target

Environment The environment is the shared, observable setting in which the available behaviors act and cooperate. It provides an abstract account of the observable preconditions and effect of actions. In general, however, we have incomplete information about the actual preconditions and effects of actions; thus, we allow the observable environment to be *non-deterministic* in general. In this way, we manage to formalize the incomplete information we have about the actual world through the non-determinism of the environment.

Formally, an environment is a non-deterministic, finite transition system $\mathcal{E} = \langle \mathcal{A}, E, e_0, \rho \rangle$, where:

- $\mathcal{A}$ is a finite set of actions;
- E is a finite set of environment states;
- $e_0 \in E$ is the initial state;
- $\rho \subseteq E \times \mathcal{A} \times E$ is the transition relation among states: $\langle e, a, e' \rangle \in \rho$, or $e \xrightarrow{a} e'$ in $\mathcal{E}$, denotes that action a performed in state e may lead the environment to a successor state e'.

To better illustrate the concept of environment (and also the concepts of behaviors and target, that will be described later in this document), we will refer to the block-world painting domain described in [Sardiña, Patrizi, and De Giacomo, 2008], in which there are three robot arms able to *prepare*, *clean*, *paint*, and *dispose* of blocks. The blocks can be painted or cleaned only after being prepared, and after being painted they need to be disposed to proceed with another unpainted block. If a block is dirty, it has to be cleaned before it can be painted. Both the cleaning and the painting actions consume resources, respectively water and paint, and thus may need to refill by executing the *recharge* action at any time.

In this example, the environment (figure 2.3) is composed of four states $E = \{e_1, e_2, e_3, e_4\}$, with e_1 being the initial state. The environment reflects the possible need to refill water or paint at any time by allowing to execute the *recharge* action in each of its states. The only action that causes the environment to evolve from e_1 to e_2 is *prepare*, and this models the requirement to prepare the block before its processing. While in the state e_2 there are two possibilities: if the block is dirty, it has to be cleaned before painting, and this causes the environment to evolve from state e_2 to state e_3, where later it can be painted and then disposed (evolving to state e_4); otherwise, if the block is not dirty, it can be directly painted and disposed, causing the environment to return in state e_1.

Behaviors A behavior models the logic of some available device or component, or a program for an agent. This program leaves the selection of the next action to be performed to the agent itself: at each step, the program presents to the agent a set of

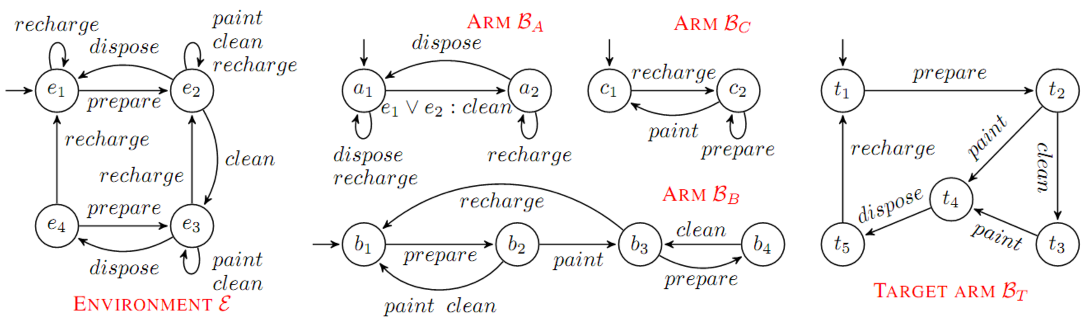


Figure 2.3: The painting arms system $\mathcal{S} = \langle \mathcal{B}_A, \mathcal{B}_B, \mathcal{B}_C, \mathcal{E} \rangle$ and the target arm $\mathcal{B}_t$

available actions, the agent selects one of them, the action is executed, and the loop is repeated.

Behaviors are intended to be executed in the environment; hence, we give them the ability of testing conditions (i.e., guards) on the environment when needed.

Formally, a behavior over an environment $\mathcal{E} = \langle \mathcal{A}, E, e_0, \rho \rangle$ is a tuple $\mathcal{B} = \langle B, B_0, \mathcal{G}, \sigma, \mathcal{F} \rangle$, where:

- B is the finite set of behavior's states;
- $B_0 \in B$ is the initial state;
- $\mathcal{G}$ is a set of *guards*, each of which is a boolean function $g : E \mapsto \{\texttt{true}, \texttt{false}\}$;
- $\sigma \subseteq B \times \mathcal{G} \times \mathcal{A} \times B$ is the behavior's tranisition relation among states: $\langle b, g, a, b' \rangle \in \sigma$, or $b \xrightarrow{g,a} b'$ in $\mathcal{B}$, denotes that the action a performed in behavior state b, when the environment is in a state such that $g(e) = \texttt{true}$, may lead the behavior to a successor state b';
- $\mathcal{F} \subseteq B$ is the set of final states of the behavior, i.e., the states in which the behavior may stop executing, but does not necessarily have to.

Behaviors are, in general, *non-deterministic*, in the sense that, given a state and an action, there could be more than one transaction whose guard function evaluates to $\texttt{true}$; as a consequence of that, the execution of the action does not give any certainty about the resulting state, i.e., the execution of the same action from the same state could result in different next states at different times.

The block-world domain presents the three robot arms as available behaviors (figure 2.3). Arm $\mathcal{B}_A$ is able to *clean* and *dispose* blocks; arm $\mathcal{B}_B$ can *prepare*, *clean* and *paint* blocks; arm $\mathcal{B}_C$ is up to *paint* and *prepare* blocks. The arm $\mathcal{B}_A$ has a guard on *clean*, meaning that it can perform that action if and only if a certain condition holds (in this case, if the environment in either in state e_1 or state e_2); the arm $\mathcal{B}_B$ presents a non-deterministic action, namely *paint*: since, as we said before, this action consumes a resource, and the way arm $\mathcal{B}_B$ stores paint is internal to its behavior, we cannot predict whether, after painting, the paint tank is empty and thus needs to be refilled (state b_3) or there is still some paint left that can be used to paint more blocks (state b_1). Finally, arm $\mathcal{B}_C$ is more conservative, in the sense that it has a smaller tank capacity able to hold the paint needed for painting just one block, so it has to recharge each time. All the behaviors can halt their execution when they reach their initial states.

Target The target behavior is a *deterministic* behavior over $\mathcal{E}$, and it represents the fully controllable desired behavior to be obtained through the available behaviors. It is important to notice that the target behavior acts like a virtual component, i.e., a component that does not exist in reality, but whose behavior is the one that the user of the system desires to obtain from the collection of the available behaviors as a whole.

In the block-world example (figure 2.3), the target behavior we want to achieve is to initially prepare a block, clean it if it is dirty, then proceed with the painting and the disposal of the block, and finally start again with a new block.

2.5.2 The Composed System

Enacted Behavior When a behavior performs an action, it causes both the environment and the behavior itself to evolve to a successor state. We can visualize the effect of the execution of an action on these two elements by defining a new abstract transition system that is called *enacted behavior*, which combines these two components as if they were a unique behavior.

Formally, given a behavior $\mathcal{B} = \langle B, b_0, \mathcal{G}, \sigma, \mathcal{F} \rangle$ over an environment $\mathcal{E} = \langle \mathcal{A}, E, e_0, \rho \rangle$, the enacted behavior of $\mathcal{B}$ over $\mathcal{E}$ is a tuple $\mathcal{T}_\mathcal{B} = \langle S, \mathcal{A}, s_0, \delta \rangle$, where:

- $S = B \times E$ is the *finite* set of states of $\mathcal{T}_\mathcal{B}$ (given a state $s = \langle b, e \rangle$, we denote b by $beh(s)$ and e by $env(s)$);
- $\mathcal{A}$ is the set of actions in $\mathcal{E}$;
- $s_0 \in S$, with $beh(s_0) = b_0$ and $env(s_0) = e_0$, is the initial state of $\mathcal{T}_\mathcal{B}$;
- $\delta \subseteq S \times \mathcal{A} \times S$ is the enacted transition relation, where $\langle s, a, s' \rangle \in \delta$, or $s \xrightarrow{a}$ in $\mathcal{T}_\mathcal{B}$, if and only if:

  (i) $env(s) \xrightarrow{a} env(s')$ in $\mathcal{E}$;

  (ii) $beh(s) \xrightarrow{a,g} beh(s')$ in $\mathcal{B}$, with $g(env(s)) = \texttt{true}$ for some $g \in \mathcal{G}$.

The enacted behavior $\mathcal{T}_\mathcal{B}$ is technically the *synchronous product* of the behavior and the environment, and represents the evolution of the behavior $\mathcal{B}$ as if it were to act alone in the environment. It is important to observe that there could be some states in the enacted behavior that may be not reachable from the initial state s_0: these state can be reached only when other behaviors are also acting in the environment.

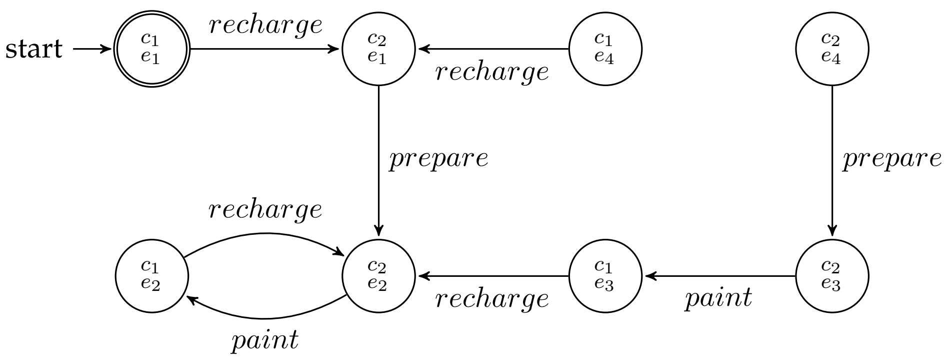


Figure 2.4: The enacted behavior of arm $\mathcal{B}_C$

The System A system $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$ is formed by an environment $\mathcal{E}$ and n predefined non-deterministic behaviors $\mathcal{B}_i$ over $\mathcal{E}$, called the *available behaviors*. A system configuration is a tuple $c = \langle s_1, \ldots, s_n, e \rangle$ denotes a snapshot of the system: behavior $\mathcal{B}_i$, with $i \in \{1, \ldots, n\}$, is in state s_i and the environment $\mathcal{E}$ is in state e.

All the available behaviors in a system are meant to act concurrently, in an interleaved fashion, in the same environment. Similarly to what we have done for the enacted behaviors, we can refer to the behavior that emerges from the joint execution of the available behaviors in the environment by defining an new abstract transition system called *enacted system behavior*.

Formally, the enacted system behavior of a system $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$, where $\mathcal{E} = \langle \mathcal{A}, E, e_0, \rho \rangle$ and $\mathcal{B}_i = \langle B_i, b_{i0}, \mathcal{G}_i, \sigma_i, \mathcal{F}_i \rangle$ for $i \in \{1, \ldots, n\}$, is the tuple $\mathcal{T}_\mathcal{S} = \langle S_\mathcal{S}, \mathcal{A}, \{1, \ldots, n\}, s_{\mathcal{S}_0}, \delta_\mathcal{S} \rangle$, where:

- $S_\mathcal{S} = B_1 \times \cdots \times B_n \times E$ is the finite set of states of $\mathcal{T}_\mathcal{S}$ (when $s_\mathcal{S} = \langle b_1, \ldots, b_n, e \rangle$, we denote b_i by $beh_i(s_\mathcal{S})$ for $i \in \{1, \ldots, n\}$, and e by $env(s_\mathcal{S})$);
- $\mathcal{A}$ is the set of actions in $\mathcal{E}$;
- $s_{\mathcal{S}_0} \in S_\mathcal{S}$, with $beh_i(s_{\mathcal{S}_0}) = b_{i0}$ for $i \in \{1, \ldots, n\}$ and $env(s_{\mathcal{S}_0}) = e_0$, is the initial state for $\mathcal{T}_\mathcal{S}$;
- $\delta_\mathcal{S} \subseteq S_\mathcal{S} \times \mathcal{A} \times \{1, \ldots, n\} \times S_\mathcal{S}$ is the transition relation for $\mathcal{T}_\mathcal{S}$, where $\langle s_\mathcal{S}, a, k, s'_\mathcal{S} \rangle \in \delta_\mathcal{S}$, or $s_\mathcal{S} \xrightarrow{a,k} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$, if and only if:
  - (i) $env(s_\mathcal{S}) \xrightarrow{a} env(s'_\mathcal{S})$ in $\mathcal{E}$;
  - (ii) $beh_k(s_\mathcal{S}) \xrightarrow{g,a} beh_k(s'_\mathcal{S})$ in $\mathcal{B}_k$, with $g(env(s_\mathcal{S})) = \texttt{true}$, for some $g \in \mathcal{G}_k$;
  - (iii) $beh_i(s_\mathcal{S}) = beh_i(s'_\mathcal{S})$, for $i \in \{1, \ldots, n\} \setminus \{k\}$.

The enacted system behavior $\mathcal{T}_\mathcal{S}$ is technically the *synchronous product* of the environment with the *asynchronous product* product of the available behaviors. It is similar to an enacted behavior except for the presence of the index k in transitions, which makes explicit that only one behavior (the one with the index k) is performing the action in the transition, while all other behaviors remain still.

2.5.3 Controllers and Compositions

Behavior Controllers A *controller* is a system component that is able to activate, stop, and resume any of the available behaviors, and to instruct them to execute an action among those allowed in their current state, taking into account also the state in which the environment is in. We assume that the controller has *full observability* on the available behaviors and the environment, i.e., it can keep track at runtime of their current states. To formally define controllers, we first need to define the following techical notions:

- a *trace* for a given enacted behavior $\mathcal{T}_\mathcal{B} = \langle S, \mathcal{A}, s_0, \delta \rangle$ is a *possibly infinite* sequence of the form $s^0 \xrightarrow{a^1} s^1 \xrightarrow{a^2} \ldots$, such that:

  (i) $s^0 = s_0$;

  (ii) $s^j \overset{a^{(j+1)}}{\longrightarrow} s^{(j+1)}$ in $\mathcal{T}_\mathcal{B}$, for all $j > 0$.

- a *history* is just a finite prefix $h = s^0 \xrightarrow{a^1} \ldots \xrightarrow{a^l} s^l$ of a trace. We denote s^l by $last(h)$, and l by $|h|$ (or $length(h)$).

Let $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$ be a system and $\mathcal{H}$ be the set of the histories of $\mathcal{T}_\mathcal{S}$. A *controller* for the system $\mathcal{S}$ is a function $P : \mathcal{H} \times \mathcal{A} \mapsto \{1, \ldots, n, u\}$ which, given a system history $h \in \mathcal{H}$ and an action $a \in \mathcal{A}$ to perform, selects a behavior to delegate a for execution. The value u (undefined) is returned in the case of irrelevants histories or actions that no behavior can perform after a given history, thus making P a total function.

Problem Statement The problem we are interested in can be stated as follows: given a system $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$ and a *deterministic* target behavior $\mathcal{B}_t$ over $\mathcal{E}$, we want to synthesize a controller P which realizes the target behavior $\mathcal{B}_t$ by suitably delegating each action requested by $\mathcal{B}_t$ to one of the available behaviors $\mathcal{B}_i$ in $\mathcal{S}$. At a first sight, a controller realizes a target if, for every trace of the enacted target behavior, at every step it returns the index of an available behavior that can perform the requested action.

Let's define more formally what *realization* means. Let $\mathcal{S} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{E} \rangle$ be a system, $\mathcal{B}_t$ a target behavior and $\mathcal{T}_t$ its correspondent enacted behavior, and P a controller for $\mathcal{S}$. We say that P *realizes* a trace $\tau = s^0 \xrightarrow{a^1} s^1 \xrightarrow{a^2} \ldots$ of the target behavior if and only if:

1. for all enacted system histories $h \in \mathcal{H}_{\tau,P}$ (defined below), if $|h| < |\tau|$ then we have that $P(h, a^{|h|+1}) = k$ and $last(h) \overset{a^{|h|+1,k}}{\longrightarrow} s'_\mathcal{S}$ in $\mathcal{T}_\mathcal{S}$ for some $s'_\mathcal{S}$. $\mathcal{H}_{\tau,P} = \bigcup_l \mathcal{H}^l_{\tau,P}$ is

the set of $\mathcal{T}_\mathcal{S}$ *histories induced by P and* τ, and it is inductively defined as follows:

(a) $\mathcal{H}^0_{\tau,P} = \{(s_{10}, \dots, s_{n0}, e_0)\}$;

(b) $\mathcal{H}^{l+1}_{\tau,P}$ is the set of $(l+1)$-length histories $h' = h \overset{a^{j+1},k^{j+1}}{\longrightarrow} s^{j+1}_\mathcal{S}$ such that:

- $h \in \mathcal{H}^j_{\tau,P}$;
- $env(s^{j+1}_\mathcal{S}) = env(s^{j+1})$;
- $P(h, a^{j+1}) = k^{j+1}$, i.e. after history h, action a^{j+1} is delegated to behavior $\mathcal{B}_{k^{j+1}}$;
- $last(h) \overset{a^{j+1},k^{j+1}}{\longrightarrow} s^{j+1}$ in $\mathcal{T}_\mathcal{S}$, i.e. behavior $\mathcal{B}_{k^{j+1}}$ is able to perform action a^{j+1}, taking into account also the current state $env(last(h))$ of the environment;

2. for each state s^l occurring in τ, if $beh(s^l)$ is a final state for $\mathcal{B}_\mathcal{T}$, then all l-length histories $h \in \mathcal{H}^l_{\tau,P}$ are such that $last(h)$ are final states for $\mathcal{B}_\mathcal{S}$.

This definition of *realization* denotes what a good controller should do: since $\mathcal{H}_{\tau,P}$ is the set of histories that the controller P could give as output while trying to realize the target trace τ, it should be able to extend the histories in $\mathcal{H}_{\tau,P}$ by choosing a behavior such that the system can evolve legally without reaching a non-valid state.

The first condition expresses the concept that the system histories are built starting from the initial state $s_{\mathcal{S}_0}$ of the enacted system, and then extended in a stepwise manner by delegating the execution of the action to the behavior selected by the controller. The second condition states that, during the execution of the sequence of actions specified by the target, if the target reaches a final state in a step, then all the behaviors should be in their final states in the same step.

If a controller is able to activate the behaviors by delegating them actions so as to always *mimic* the target behavior, then the target is realizable by the available system. Since the target behavior is a deterministic transition system, a controller P is said to realize a target behavior $\mathcal{T}_t$ if and only if it realizes *all the traces* of $\mathcal{T}_t$.

All the contollers that are a solution for the behavior composition problem are called *compositions* of $\mathcal{T}_t$ over $\mathcal{E}$.

CHAPTER 3

ADOPTED TECHNOLOGY

3.1 Unity

3.1.1 Unity engine and Angry Bots game

Unity [1] is a game engine developed and distributed by Unity Technologies, a San Francisco-based company founded in 2004 by David Helgason, Nicholas Francis, and Joachim Ante. The Unity game engine began to become widespread in 2008 when, with Apple's iPhone starting to dominate the mobile phone market, Unity was one of the first engines supporting the platform in full. Another boost in Unity's popularity came later in 2009, when the creators decided to start a feature limited version of the Unity game engine for free.

Unity's success is due to the ability to deploy a video game developed with its technology to a plethora of different platforms, such as video game consoles (Xbox 360, PlayStation 3, Wii and WiiU), mobile phones (iOS and Android), web browsers (Adobe Flash and Unity Web Player), and traditional Operating Systems (Windows, MacOS X and Linux). Another critical factor in Unity's acclaim is the support it offers

[1] `http://unity3d.com`

to independent developers, who were previously unable to create their own engine or purchase licences to adopt existing ones, leading to a sort of "democratization of game development".

Angry Bots is the name of the technical demo that was shipped with the version 3.5 of Unity. It is a simple third-person shooter game composed by a single level, in which the player is required to explore the high-tech environment, filled with enemy robot sentinels, to interact with computers to open doors, and ultimately to leave the facility unharmed after blowing it up by setting up a bomb. It is intended to be a showcase of the Unity game engine capabilities, and in fact it features captivating graphics and it is designed to run on various platforms (primarily iOS and Android, but also Windows and MacOS X).

In the development of the project for this master's thesis, we used the Angry Bots tech demo as a starting point, reusing the assets of commercial-quality provided, and we extended it exploiting the C# scripting feature offered by Unity. A screenshot of Angry Bots can be seen in figure 3.1.

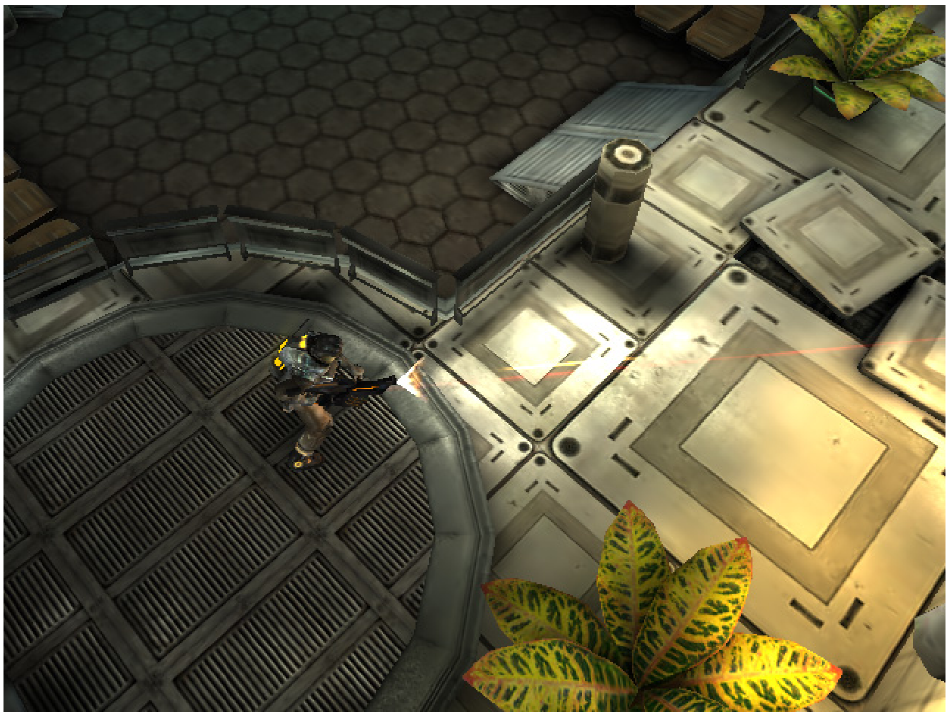


Figure 3.1: A screenshot of the Angry Bots technical demo

3.1.2 Essential Unity concepts

The Unity game engine and its built-in editor make use of some concepts that could be unfamiliar to work with for both people with no previous experience in game programming and users coming from other engines or editors. In this section, we will present the key ideas upon which the Unity framework is built, and also the interface that will constitute the workspace which the user will use in order to turn his ideas into an actual game.

The most basic entity in the Unity game engine is called a *GameObject*. By itself, a GameObject is almost worthless, because the essence and the usefulness of a GameObject is to have its characteristics defined by using *components*. A component is usually, but not always, a fragment of source code (in the Unity terminology, a *script*) that the programmer could write selecting his favorite language among C#, JavaScript and Boo (a Python derivative) and, by "attaching" components to GameObjects, the game engine will known where it should position them, how big they are, what their behavior in particular situations will look like, and so on. An example of component is the **Transform** one, which comes built-in when a new GameObject is created, and defines the location, the rotation and the scale of the object it is attached to. Besides GameObjects, other important basic concepts in Unity are represented by *scenes*, that correspond to the levels of the game or other screens to be displayed (e.g., menus) and *prefabs*, that are containers for scripts, models, textures, and components used to form a "template" object ready to be recycled. Collectively, all the resources involved in a Unity project are referred as *assets*.

The Unity editor interface is composed of several dockable panels, that come in some predefined layouts but that can be freely rearranged by the user. The sections in which the editor is divided are:

- The **Scene** panel, in which the user can navigate and modify visually the game world, and also drag assets from the Project panel to make them become active GameObjects;

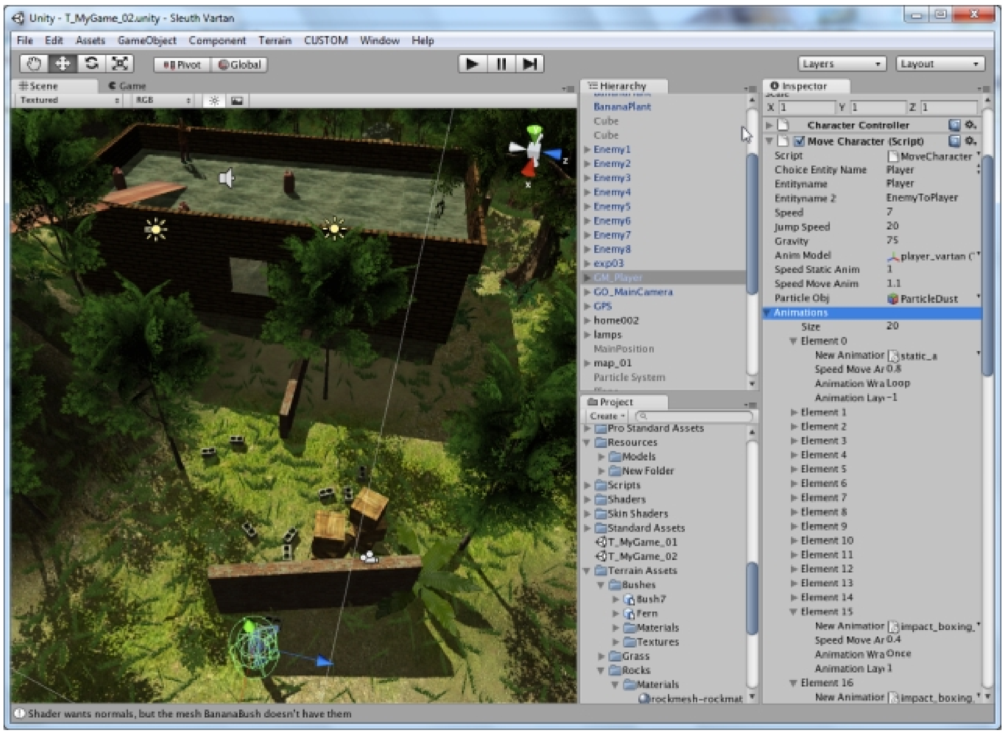
- The **Hierarchy** panel, which contains a list of all the GameObjects present in the currently opened scene, ordered alphabetically by name;

- The **Inspector** panel, that shows the components attached to the selected GameObject, and also allows an easy modification of all the variables involved in the script;

- The **Project** panel: this section contains a direct view of the *Assets* folder of the project, and also permits the creation of new assets directly within the Unity interface;

- The **Game** panel, that acts as a fully featured test of the game, also supporting various screen aspect ratios.

A screenshot of the Unity interface is given in figure 3.2. Here, on the left side there is the **Scene** panel; on the right side the **Inspector** panel; in the middle, the **Hierarchy** panel is placed above the **Project** panel; and the **Game** panel is not shown.



```
X 1          Y 1          Z 1
Character Controller
Move Character (Script)
Script                  MoveCharacter
Choice Entity Name      Player
Entityname              Player
Entityname 2            EnemyToPlayer
Speed                   7
Jump Speed              20
Gravity                 75
Anim Model              player_vartan (
Speed Static Anim       1
Speed Move Anim         1.1
Particle Obj            ParticleDust
Animations
    Size                20
  Element 0
      New Animatior     static_a
      Speed Move Ar     0.8
      Animation Wra     Loop
      Animation Lay     -1
  Element 1
  Element 2
  Element 3
  Element 4
  Element 5
  Element 6
  Element 7
  Element 8
  Element 9
  Element 10
  Element 11
  Element 12
  Element 13
  Element 14
  Element 15
      New Animatior     impact_boxing_
      Speed Move Ar     0.4
      Animation Wra     Once
      Animation Lay     1
  Element 16
      New Animatior     impact_boxing_
```

Shader wants normals, but the mesh BananaBush doesn't have them

Figure 3.2: The Unity Editor interface

3.1.3 NPC Reactive Behaviors as Finite State Machines

As we said previously, the visually appealing level of the *Angry Bots* technical demo is filled with enemy bots whose main objective is to prevent the player from reaching the most inner area of the facility, even resorting to an heroic auto-destruction if things go particularly bad (or good, from the player's point of view).

Analyzing the source code of the scripts responsible for bringing these bots to life, we can notice that the programmers at Unity Technologies decided to implement the reactive behavior of the non-player characters using a loosely-defined finite state machine. For example, the script named `MechAttackMoveController.js`, located in the `Assets\Scripts\AI` folder of the project, contains the following source code:

```
private var inRange : boolean = false;

function Update () {
  // Calculate the direction from the player to this character
  var playerDirection : Vector3 = (player.position - character.position);
  playerDirection.y = 0;
  var playerDist : float = playerDirection.magnitude;
  playerDirection /= playerDist;

  if (inRange && playerDist > targetDistanceMax)
    inRange = false;
  if (!inRange && playerDist < targetDistanceMin)
    inRange = true;

  if (inRange)
    motor.movementDirection = Vector3.zero;
  else
    motor.movementDirection = playerDirection;

}
```

The `Update()` function is called by the Unity game engine once per frame, thus it should contain all the decision-making logic that controls the non-player character perception of the game world and the related actions. The most common way to represent a finite state machine in code is to encode all the possible states employing a finite set of enumerated values, and having the execution flow of the `Update()` function diversified based on the current state; an example of implementation of this concept is shown in the following source code fragment:

```
var states = {
  IDLE: 0,
  ACTIVE: 1,
  ATTACKING: 2,
};

var currentState = states.ACTIVE;

function Update () {
  switch(currentState) {
    case states.IDLE:
      // Code for the IDLE state
      break;
    case states.ACTIVE:
      // Code for the ACTIVE state
      break;
    case states.ATTACKING:
      // Code for the ATTACKING state
      break;
    default:
      break;
  }
}
```

Even if the section of the code showed before doesn't follow strictly the structure depicted above, the `inRange` boolean variable effectively divides the flow of execution of the `Update()` function into two different branches based on its value; thus, we can consider the reactive behavior of the non-player character as if it was regulated by a virtual finite state machine composed by two states (let's call them `IDLE` and `ACTIVE`), and whose transitions between the two states are triggered by the change of value of the `inRange` variable:

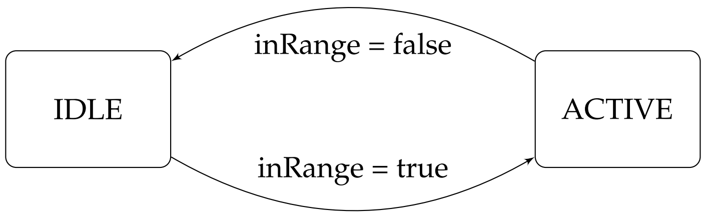


Figure 3.3: The virtual state machine for the NPCs in Angry Bots

3.1.4 A* Pathfinding Project

The **A* Pathfinding Project** [2], developed by Aron Granberg, is a wonderful and extremely well-documented library, written in C#, for addressing one of the features present in the Professional edition of Unity but not in the Free version (used in the development of this master thesis), namely pathfinding. Having in mind to visualize the behavior composition concept by means of a collection of robots patrolling a crisp environment, pathfinding was at the same time one of the essential building blocks for the project and the first implementation problem to solve; by relying on the efficient and easily understandable code provided by this library, we have been able to start from scratch and build the first working version of the project with the robots making their way to various destinations in just a handful of days.

The A* Pathfinding Project library is able to work on different kinds of graph, such as navmeshes, grid graphs, and point graphs; it also offers the opportunity to extend its functionalities by defining your own graph type, as we did with the `Pathfinding Graph` employed in the development of this master's thesis. Also, it is capable of generating automatically the navmesh graph from the world geometry, doing in a few seconds a job that would require some hours to do manually. Other interesting features of this great software component include the availability on a broad range of platforms (basically all the platforms that support Unity, with the exception of Adobe Flash), and the ability of dealing with a great number of agents concurrently via the usage of multi-threading programming.

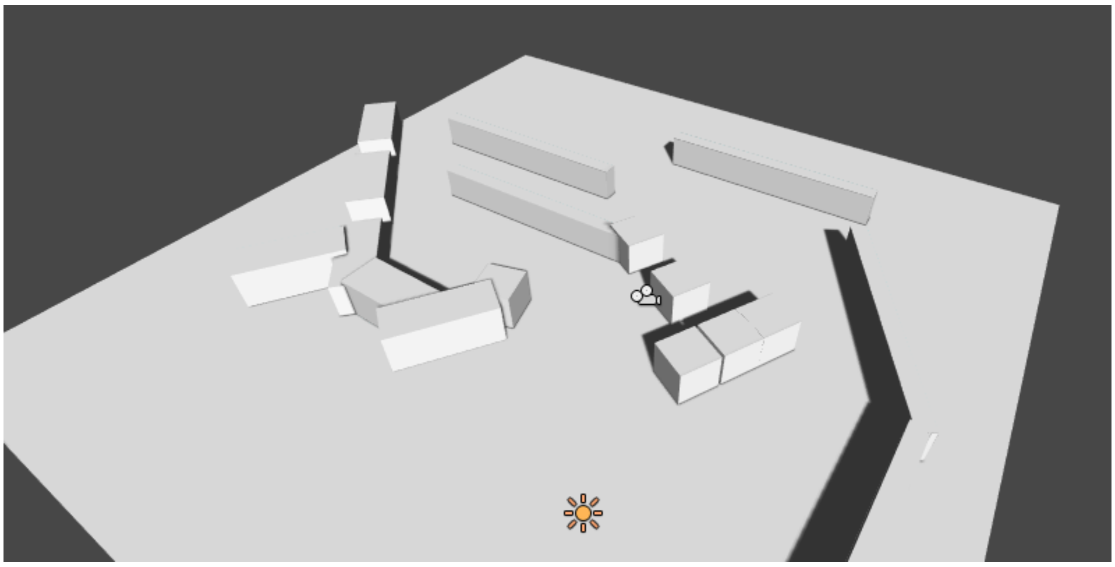
[2] `http://arongranberg.com/astar/`

Getting started with A* Pathfinding Project

- The central script of the A* Pathfinding Project is the script 'astarpath.cs', it acts as a central hub for everything else.
- In the AstarPath inspector you create all graphs and adjust all settings.
- There should always be one (always one, no more) astarpath.cs component in a scene which uses pathfinding.
- The astarpath.cs script can be found at Components → Pathfinding → Pathfinder
- The second most important component is the 'Seeker.cs' component; a Seeker component should be attached to every GameObject which uses pathfinding (e.g., all AIs).
- The Seeker component handles path calls for one unit and post processes the paths. The Seeker is not needed, but it makes pathfinding easier.
- Lastly there are the modifier scripts (e.g SimpleSmoothModifier.cs). Modifiers post-processes paths to smooth or simplify them, if a modifier is attached to the same GameObject as a Seeker it will post-process all paths that Seeker handles.

New Scene

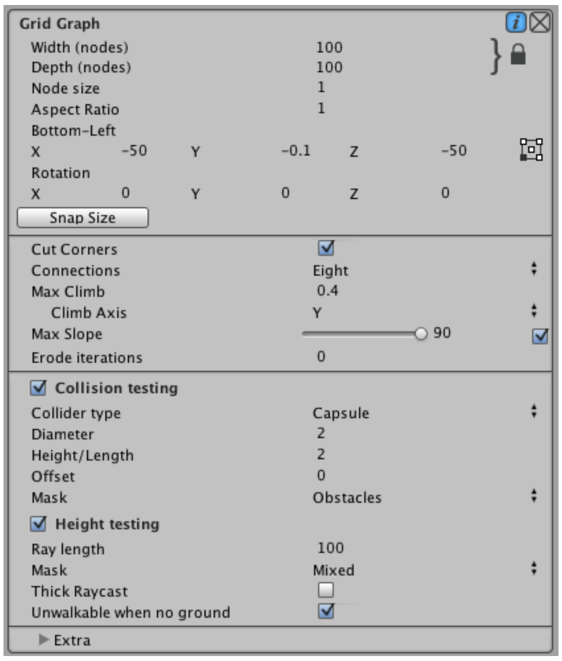
Create a new scene, name it "PathfindingTest". Now let's create something which an AI could walk on and something for it to avoid: add a plane to the scene, place it in the scene origin (0,0,0) and scale it to 10,10,10. Create a new layer (Edit → Project Settings → Tags) named "Ground" and place the plane in that layer. Now create some cubes of different scales and place them on the plane, these will be obstacles which the AI should avoid. Place them in a new layer named "Obstacles". Your scene should now look something like this:



Adding A*

Now we have ground for an AI to stand on and obstacles for it to avoid. So now we are going to add the A* Pathfinding System to the scene to enable pathfinding. Create a new GameObject, name it "A*", add the "AstarPath" component to it (Components → Pathfinding → Pathfinder). The AstarPath inspector is divided into several parts. The two most important is the Graphs area and the Scan button at the bottom. The Graphs area holds all the graphs in your scene, you may have up to 16 but usually 1 or 2 will be sufficient. A single graph is usually to be preferred. If you open the Graphs area by clicking on it you will see a list of graphs which you can add. The two main ones are the Grid Graph, which generates nodes in a grid pattern, and the Navmesh Graph, which takes a mesh as the walkable area. The Scan button is for updating the graphs, this is also done on startup (unless the startup is cached) and some graphs will do it automatically when changing the graph settings and the scanning won't cause any lag. There is also a shortcut for scanning: Cmd+Alt+S (Mac) or Ctrl+Alt+S (Windows).

For this tutorial we will create a Grid Graph; after adding it, click on the new Grid Graph label to bring up the graph inspector.



Grid Graph

Width (nodes)	100				
Depth (nodes)	100				
Node size	1				
Aspect Ratio	1				
Bottom-Left					
X	-50	Y	-0.1	Z	-50
Rotation					
X	0	Y	0	Z	0

Snap Size

Cut Corners	✓
Connections	Eight
Max Climb	0.4
Climb Axis	Y
Max Slope	90 ✓
Erode iterations	0

✓ Collision testing

Collider type	Capsule
Diameter	2
Height/Length	2
Offset	0
Mask	Obstacles

✓ Height testing

Ray length	100
Mask	Mixed
Thick Raycast	☐
Unwalkable when no ground	✓

▶ Extra

As the name implies, the GridGraph will generate a grid of nodes, width*depth. A grid can be positioned anywhere in the scene and you can rotate it any way you want. The Node Size variable determines how large a square/node in the grid is, for this tutorial you can leave it at 1, so the nodes will be spaced 1 unit apart. The position needs to be changed though. Switch to bottom-left in the small selector to the right of the position field (currently named "Center"), then enter (-50,-0.1,-50). The -0.1 is to avoid floating point errors, in our scene the ground is at Y=0, if the graph was to have position Y=0 too, we might get annoying floating point errors when casting rays against it for example (like the height check does). To make the grid fit our scene we need to change the width and depth variables, set both to 100 in this case. You can see that the grid is correctly positioned by the white bounding rectangle in the scene view which should now be enclosing the plane exactly.

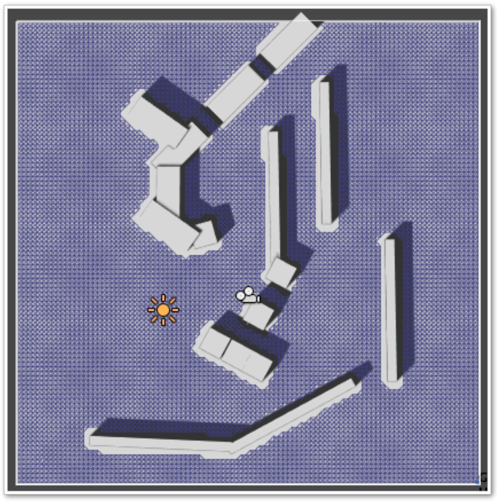
Height Testing

In order to place the nodes at their correct height, the A* system creates and fires off a set of rays against the scene to see where they hit. That's the Height Testing settings. A ray, optionally thick (as opposed to a line), is fired from [Ray Length] units above the grid downwards, a node is placed where it hits. If it doesn't hit anything, it is either made unwalkable (if the Unwalkable When No Ground variable is toggled) or the node is placed at Y=0 relative to the grid if it is set to false. We need to change the mask used, currently it includes everything, but that would include our obstacles as well, and we don't want that. So set the Mask to only include the "Ground" layer which we created earlier.

Collision Testing

When a node has been placed, it is checked for walkability; this can be done with a Sphere, Capsule or a Ray. Usually a capsule is used with the same diameter and height as the AI character which is going to be walking around in the world, preferably with some margin though. Our AI will have the standard diameter and height of 1 and 2 world units respectively, but we will set the diameter and height for the collision testing to 2 and 2 to get some margin. Next, to make the system aware of the obstacles we placed, we need to change the mask for the Collision Testing, this time set it to contain only the "Obstacles" layer as we wouldn't want our ground to be treated as an obstacle.

Now everything should be set up correctly to scan the graph. Press Scan. Wait a fraction of a second, and the grid should be generated (if you have done everything correctly, that is, compare your settings to the image below, also check that Show Graphs is true).



Adding the AI

After generating the grid, let's add an AI to play around with. Create a Capsule and add the Character Controller component to it, also place it somewhere visible on the plane. Add the Seeker component to the AI; this script is a helper script for calling pathfinding from other scripts, it can also handle path modifiers which can, for example, smooth the path or simplify it using raycasts. We are going to write our own, really simple script for moving the AI, so open your favourite script-editor and follow.

Moving Stuff Around

The call to the Seeker is really simple, three arguments: a start position, an end position and a callback function (must be in the form "void SomeFunction (Path p)"):

```
StartPath (Vector3 start, Vector3 end, OnPathDelegate callback = null) : Path
```

So let's start our script with a simple script calling the pathfinder at startup:

```
using UnityEngine;
using System.Collections;
using Pathfinding;

public class AstarAI : MonoBehaviour {
  public Vector3 targetPosition;

  public void Start () {
    //Get a reference to the Seeker component we added earlier
    Seeker seeker = GetComponent<Seeker>();
    //Start a new path to the targetPosition, return the result to the
        OnPathComplete function
    seeker.StartPath (transform.position, targetPosition, OnPathComplete);
  }

  public void OnPathComplete (Path p) {
    Debug.Log ("We got a path back. Did it have an error? " + p.error);
  }
}
```

Save it to a file in your project named AstarAI.cs and add the script to the AI GameObject. In the inspector, change Target Position to something like (-20,0,22). This is the position the AI will try to find a path to now. Press Play. You should get the log message and also the path should show in the scene view as a green line (the Seeker component draws the latest path using Gizmos). If you do not see a green line, make sure that the checkbox "Show Gizmos" on the Seeker component is checked. More recent Unity versions also depth-test gizmos, so it might be hidden under the ground; to disable the depth-testing, click the Gizmos button above the scene view window and uncheck the "3D Gizmos" checkbox. In case you get an error, make sure that the Seeker component is attached to the same GameObject as the AstarAI script. If you still get an error, the target position might not be reachable, so try to change it.

What happens is that first the script calls the Seeker's StartPath function. The seeker will then create a new Path instance and then sent it forward to the AstarPath script (the Pathfinder component you added before). The AstarPath script will put the path in a queue. When available, the script will then process the path by searching the grid, node by node until the end node is found. Once calculated, the path is returned to the Seeker which will post-process it (if any modifiers are attached), and then the Seeker will call the callback function specified in the call. The callback is also sent to Seeker.pathCallback, which you can register to if you don't want to specify a callback every time you call StartPath:

```
//OnPathComplete will be called every time a path is returned to this seeker
seeker.pathCallback += OnPathComplete;
//So now we can omit the callback parameter
seeker.StartPath (transform.position,targetPosition);
```

When we get the calculated path back, how can we get information from it? A Path instance contains two arrays related to that. Path.vectorPath is a Vector3 array which holds the path, this array will be modified if any smoothing is used, it is the recommended way to get a path. Secondly,there is the Path.path array, which is an array of Node elements; it holds all the nodes the path visited, which can be useful to get additional info on the traversed path. First though, you should always check path.error: if that is true, the path has failed for some reason. Path.errorLog will have more information on what went wrong in case path.error is true.

To expand our AI script, let's add the CharacterController to the AI gameObject.

```
using UnityEngine;
using System.Collections;
using Pathfinding;

public class AstarAI : MonoBehaviour {
  //The point to move to
  public Vector3 targetPosition;
  private Seeker seeker;
  private CharacterController controller;
  //The calculated path
  public Path path;
  //The AI's speed per second
  public float speed = 100;
  //The max distance from the AI to a waypoint for it to continue to the next
       waypoint
  public float nextWaypointDistance = 3;
  //The waypoint we are currently moving towards
  private int currentWaypoint = 0;

  public void Start () {
    seeker = GetComponent<Seeker>();
    controller = GetComponent<CharacterController>();
    //Start a new path to the targetPosition, return the result to the
        OnPathComplete function
    seeker.StartPath (transform.position,targetPosition, OnPathComplete);
  }

  public void OnPathComplete (Path p) {
    Debug.Log ("We got a path back. Did it have an error? "+p.error);
    if (!p.error) {
      path = p;
      //Reset the waypoint counter
      currentWaypoint = 0;
    }
  }

  public void FixedUpdate () {
    if (path == null) {
      //We have no path to move after yet
      return;
    }
    if (currentWaypoint >= path.vectorPath.Count) {
      Debug.Log ("End Of Path Reached");
      return;
    }
    //Direction to the next waypoint
    Vector3 dir = (path.vectorPath[currentWaypoint]-transform.position).
        normalized;
    dir *= speed * Time.fixedDeltaTime;
    controller.SimpleMove (dir);
    //Check if we are close enough to the next waypoint
    //If we are, proceed to follow the next waypoint
    if (Vector3.Distance (transform.position,path.vectorPath[currentWaypoint
        ]) < nextWaypointDistance) {
      currentWaypoint++;
      return;
    }
  }
}
```

If you press play now, the AI will follow the calculated path. What the code does is: in FixedUpdate, it gets the normalized direction towards the next waypoint, moves towards it a bit, then it checks if it is close enough to continue to the next waypoint. In this example, this is done by simply by incrementing the currentWaypoint index.

3.2 Representational State Transfer (REST)

3.2.1 REST principles

The terms “representational state transfer” and “REST” were introduced and defined for the first time by Roy Thomas Fielding in his PhD thesis [Fielding, 2000]. In the dissertation, Fielding defines a methodology to talk about *architectural styles*, which are high-level, abstract patterns that express the core ideas behind an architectural approach and come with a set of *constraints* that define it. The REST architectural style is described through a list of six constraints to be applied on the architecture that Fielding defines only theoretically, leaving the implementation of the individual components free to design. The constraints upon which REST is built are:

- Client-server: there is a clear separation of concerns between clients and servers of REST that allows the two components to be replaced and evolve independently each other, as long as the uniform interface between them is not altered;

- Stateless: each request from the client to the server must contain all of the information necessary to understand the request, and cannot rely on any stored context on the server;

- Cacheable: the data within a response to a request must be implicitly or explicitly labeled as cacheable or not. If a response is cacheable, the client can store and reuse that response for later, equivalent requests;

- Layered system: the architecture may be composed of hierarchical layers such that each component cannot "see" beyond the immediate layer with which they are interacting;

- Uniform interface: this is the central feature that distinguishes REST from other architectural styles. The uniform interface between clients and servers simplifies the architecture and decouples the implementations from the services they provide, encouraging independent evolvability for each part;

- Code-on-demand: REST allows the clients to extend their functionality by the transfer from the server of executable code, in the form of *compiled components* (such as Java applets) or *client-side scripts* (such as JavaScript). This is the only REST constraint considered *optional*.

More practically, REST is a set of principles that define how the Web standards, such as HTTP or URIs, are supposed to be used for the realization of Web services. The key principles defined in REST are:

- Identification of *resources*: a resource is the key abstraction of information in REST, and every information that is meaningful should get an ID. On the Web, the predominant concept for IDs is represented by Uniform Resource Identifiers (*URIs*); since URIs make up a global namespace, using URIs to identify resources means they get a unique, global ID;
- Hypermedia as the engine of the application state: behind this intimidating description lies the idea of *links*, that can be used to address further information to be retrieved. Another important aspect to the hypermedia principle is the *state* part of the application: the links that the server provides allows the client to move the application from one state to another by following a link;
- Manipulation of resources through representations: a single resource can be handled in multiple representations and, using HTTP content negotiation, clients can ask for the resource to be delivered in the most suitable representation for their needs (e.g., HTML or XML);
- Self-descriptive messages: each client request and server response is a message taken from a constrained set of message types fully understood by both clients and servers. These standard messages are called *HTTP verbs* and comprise GET, HEAD, OPTIONS, PUT, POST and DELETE; the meaning of these methods, along with guarantees about their behaviors (e.g., *idempotence* of the GET method) and meanings of the messages metadata (e.g., HTTP *headers*), are described in the HTTP 1.1 specification.

3.2.2 REST in practice

The most known application of REST is the World Wide Web, in which text, graphics, audio, video and other kinds of media are stored across a network and interconnected through hyperlinks. In the case of web services, the World Wide Web also acts as the distributed hypermedia system in which they are deployed. In this scenario, HTTP acts as both the messaging system needed to send request and receive responses upon which interactions are built, and the transport protocol used to transfer resources of interest. The payload of HTTP messages can carry information according to the MIME system, and HTTP provides a set of response status codes to inform the requester whether a request succeeded and, if not, why.

In practice, a resource is an informational item that has hyperlinks to it. Hyperlinks use URIs to do the linking. The concept of a resource is remarkably broad but, at the same time, impressively simple and precise. As Web-based informational items, resources are pointless unless they have at least one representation. In the Web, representations are MIME-typed. The most common type of resource representation is probably still `text/html`, but nowadays resources tend to have multiple representations. Resources have state, and a useful representation must capture a resource's state.

In a RESTful request targeted at a resource, the resource itself remains on the service machine. The requester typically receives a representation of the resource if the request succeeds. It is the representation that transfers from the service machine to the requester machine. In different terms, a RESTful client issues a request that involves a resource, for instance, a request to read the resource. If this read request succeeds, a typed representation (for instance, text/html) of the resource is transferred from the server that hosts the resource to the client that issued the request. The representation is a good one only if it captures the resource's state in some appropriate way.

In summary, RESTful web services require not just resources to represent, but also client-invoked operations on such resources. At the core of the RESTful approach is the insight that HTTP is not just a transport protocol, but it exposes an API through the usage of *verbs* (or *methods*), that can be used to request CRUD operations (Create, Read, Update, and Delete) on remotely stored resources. A key guiding principle of the RESTful style is to respect the original meanings of the HTTP verbs: for example, a GET request should not cause side effects (or, technically speaking should be *idempotent*) because the GET method expresses a *read* operation rather than a *create*, *delete* or *update* operation. Each HTTP request includes a verb to indicate which CRUD operation should be performed on the resource. A good representation is precisely one that matches the requested operation and captures the resource's state in some appropriate way.

HTTP verb	**Meaning in CRUD terms**
POST	*Create* a new resource from the request data
GET	*Read* a resource
PUT	*Update* a resource from the request data
DELETE	*Delete* a resource

Table 3.1: HTTP verbs and CRUD operations

In RESTful services, URIs act as identifying nouns and HTTP methods act as verbs that specify operations on the resources identified by these nouns. The HTTP verb comes first, then the URI, and finally the requester's version of HTTP. This URI is, of course, a URL that locates the web service. Table 3.2 uses simplified URIs to summarize the intended meanings of HTTP/URI combinations.

HTTP verb/URI	**Intended CRUD meaning**
POST emps	Create a new employee from the request data
GET emps	Read a list of all employees
GET emps?id=27	Read a singleton list of employee 27
PUT emps	Update the employee list with the request data
DELETE emps	Delete the employee list
DELETE emps?id=27	Delete employee 27

Table 3.2: Sample HTTP verbs/URI pairs

These verb/URI pairs are terse, precise, and uniform in style. The pairs illustrate that RESTful conventions can yield simple, clear expressions about which operation should be performed on which resource. The POST and PUT verbs are used in requests that have an HTTP body; hence, the request data are housed in the HTTP message body. The GET and DELETE verbs are used in requests that have no body; hence, the request data are sent as query string entries.

HTTP also has standard response codes, such as 404 to signal that the requested resource could not be found, and 200 to signal that the request was handled successfully.

In short, HTTP provides request verbs and MIME types for client requests and status codes (and MIME types) for service responses. Modern browsers generate only GET and POST requests. Moreover, many applications treat these two types of requests interchangeably. For example, Java `HttpServlets` have callback methods such as `doGet` and `doPost` that handle GET and POST requests, respectively. Each callback has the same parameter types, `HttpServletRequest` (the key/value pairs from the requester) and `HttpServletResponse` (a typed response to the requester). It is common to have the two callbacks execute the same code (for instance, by having one invoke the other), thereby conflating the original HTTP distinction between read and create. Table 3.3 reports some of the most common status codes included in HTTP responses.

HTTP status code	Official reason	Meaning
200	OK	Request OK.
400	Bad request	Request malformed.
403	Forbidden	Request refused.
404	Not found	Resource not found.
405	Method not allowed	Method not supported.
415	Unsupported media type	Content type not recognized.
500	Internal server error	Request processing failed.

Table 3.3: Sample HTTP verbs/URI pairs

In general, status codes in the range of 100-199 are informational; those in the range of 200-299 are success codes; codes in the range of 300-399 are for redirection; those in the range of 400-499 signal client errors; and codes in the range of 500-599 indicate server errors.

3.2.3 JSON

JSON[3] is a text-based, open standard designed for human-readable data interchange. It is derived from the JavaScript scripting language for representing simple data structures and associative arrays, called objects. Despite its relationship to JavaScript, it is

[3]`http://www.json.org/`

language-independent, with parsers available for many languages.

The JSON format was originally specified by Douglas Crockford, and is described in RFC 4627. The official Internet media type for JSON is `application/json`. The JSON format is often used for serializing and transmitting structured data over a network connection. It is used primarily to transmit data between a server and web application, serving as an alternative to XML.

JSON's basic types are:

- Number (double precision floating-point format in JavaScript, generally depends on implementation)
- String (double-quoted Unicode, with backslash escaping)
- Boolean (true or false)
- Array (an ordered sequence of values, comma-separated and enclosed in square brackets; the values do not need to be of the same type)
- Object (an unordered collection of key:value pairs with the ':' character separating the key and the value, comma-separated and enclosed in curly braces; the keys must be strings and should be distinct from each other)
- `null` (empty)

The following example shows the JSON representation of an object that describes a person. The object has string fields for first name and last name, a number field for age, contains an object representing the person's address, and contains a list (an array) of phone number objects.

```
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": 10021
  },
  "phoneNumber": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "fax",
      "number": "646 555-4567"
    }
  ]
}
```

3.2.4 A REST example: the Dropbox API

Dropbox[4] is a file hosting service operated by Dropbox, Inc., that offers cloud storage, file synchronization, and client software. Dropbox allows users to create a special folder on each of their computers, which Dropbox then synchronizes so that it appears to be the same folder (with the same contents) regardless of which computer is used to view it. Files placed in this folder also are accessible through a website and mobile phone applications.

Dropbox is one of the many services on the Internet that expose a RESTful API for developers, which can take advantage of these publicly available facilities to write new clients for the original service, or even to create a completely new service by obtaining and combining data coming from different providers.

Table 3.4 reports some of the Dropbox API endpoints, by means of which the user can request the execution of some operations on the files present on the Dropbox server, such as downloading a certain file, retrieving its metadata, or searching for files that match the given search string.

For example, by sending an HTTP message, using POST or GET as the verb, to the `https://api.dropbox.com/1/search/<root>/<path>?query=queryStr`, the

[4]`http://www.dropbox.com/`

URI	Operation
GET /files/<root>/<path>	Downloads the file with path <path>.
POST /files/<root>/<path>	Uploads a file to the folder <path>.
GET /metadata/<root>/<path>	Retrieves file and folder metadata.
GET /search/<root>/<path>	Returns metadata for all files and folders whose filename contains the given search string as a substring.
POST /shares/<root>/<path>	Creates and returns a Dropbox link to files or folders users can use to view a preview of the file in a web browser.
POST /chunked_upload?param=val	Uploads large files to Dropbox in mulit-ple chunks.

Table 3.4: Dropbox REST API Example

user can search for the files or the folders present on the Dropbox server that match the search string specified in `queryStr`. The result given by the service is a JSON array, that acts as a list of metadata entries for any matching files and folder.

If we search for `.txt` in Dropbox/Public folder, we might get as output the following JSON code:

```
[
    {
        "size": "0 bytes",
        "rev": "35c1f029684fe",
        "thumb_exists": false,
        "bytes": 0,
        "modified": "Mon, 18 Jul 2011 20:13:43 +0000",
        "path": "/Public/latest.txt",
        "is_dir": false,
        "icon": "page_white_text",
        "root": "dropbox",
        "mime_type": "text/plain",
        "revision": 220191
    }
]
```

CHAPTER 4

THE ANGRY BOTS PATROLLING DOMAIN

In order to investigate the application of the behavior composition concept with respect to the problem of coordinating a set of non-playable character, we developed a simple example that can be summarized as follows: in the environment in which the Angry Bots technical demo takes place, we identified a series of 20 points of interest (named with the letters of the alphabet from A to T); we placed in this environment three non-player characters, reusing the 3D models and the textures that came with the demo, and we gave to each NPC the ability of moving between these points of interest in such a way that no robot could cover all the points of interest by itself. We also added some overlaps on the NPCs' routes, i.e., there are some points of interest that could be reached by more than one robot. The coordination we want to achieve, i.e., the target behavior, is to put the user in charge of selecting the location he is interested to control and our behavior composition system to handle which NPC will realize the request. In particular, the controller provided by our system can select the right behavior to perform the chosen action, guaranteeing that the execution will not violate the properties that make the composition valid, for all future states.

To pursue this goal, we coded a collection of scripts in C# to be attached to the `GameObject`s involved in our framework. The main classes that constitute the framework developed for the Unity visualization of the behavior composition problem are the following:

- **FiniteStateMachine**, contained in the `FiniteStateMachine.cs` script, which holds a representation of the internal behavior of each of the non-player characters, and is also able to parse it from a text-based file;
- **PathfindingGraph**, contained in the `PathfindingGraph.cs` script, which is essential to permit the A* Pathfinding Project library to find a way for reaching the point of interest selected by the user;
- **TargetBehavior**, contained in the `TargetBehavior.cs` script, which is a subclass of the FiniteStateMachine, with which shares the internal data structures but has different definitions of the methods for reading the behavior from the TGF file and writing it to the XML format;
- **ActionLookupTable**, contained in the `ActionLookupTable.cs` script, which is the big hash table in which the entries are represented by pairs <current state of the environment, next actions available>;
- **ActionLookupGenerator**, contained in the `ActionLookupGenerator.cs` script, which served as the first way of getting the ActionLookupTable populated by generating the Cartesian product of all the possible behaviors' states;
- **ControllerGUI**, contained in the `ControllerGUI.cs` script, which is responsible to query the individual non-player characters about their current state, to query the ActionLookupTable to retrieve the possible next actions from the current state of the environment, and to display them to the user in order to make him choose one to realize the target behavior;
- **MoveNPC**, contained in the `MoveNPC.cs` script, which instructs the non-player

characters on how to move to the point of interest selected by the user via the ControllerGUI

All of these classes will be introduced and explained in this section of the thesis.

4.1 FiniteStateMachine

This script contains a simple representation of the Finite State Machine that lies behind the behavior adopted by the non-player characters. A finite state machine essentially represents an available behavior as described in the behavior composition framework in Section 2.5; note that for simplicity we assume no guard conditions. It is implemented by maintaining a list of `FSMNodeWithTransitions`, which is a structure defined by us comprising the name of the state and a list of `FSMTransitions`, which in turn keep track of the name of the transition and the destination state. The script also holds in memory the current state in which the non-player character is in and the list of possible next actions, which are the transitions that have as starting node the current state.

The states and the transitions that compose the Finite State Machine are not *hard-coded*, i.e. they are not embedded directly into the source code (as they were, for example, in the code fragment we showed in the previous section), but they are built at runtime upon the invocation of the `Start()` function, that the Unity game engine calls when it initializes the `GameObject` the script is attached to: all the pieces of information the script needs to perform this task are stored into some files encoded with the **Trivial Graph Format** [1], which is a simple text-based format for describing graphs. An example of a graph formalized using this scheme is shown in figure 4.1, which describes the graph shown in Figure 4.2.

In the Angry Bots patrolling domain, the states of the finite state machine represent the various points of interest in which the non-player character can stay in or move to, and the transitions indicate the connections between these locations that are reachable by walking. The information needed by the A* Pathfinding Project to make the robots

[1] `http://en.wikipedia.org/wiki/Trivial_Graph_Format`

```
1 Node1
2 Node2
3 Node3
4 Node4
#
1 2 TransitionBetween1And2
1 3 TransitionBetween1And3
2 4 TransitionBetween2And4
3 4 TransitionBetween3And4
```

Figure 4.1: Example of graph described through the Trivial Graph Format (code)

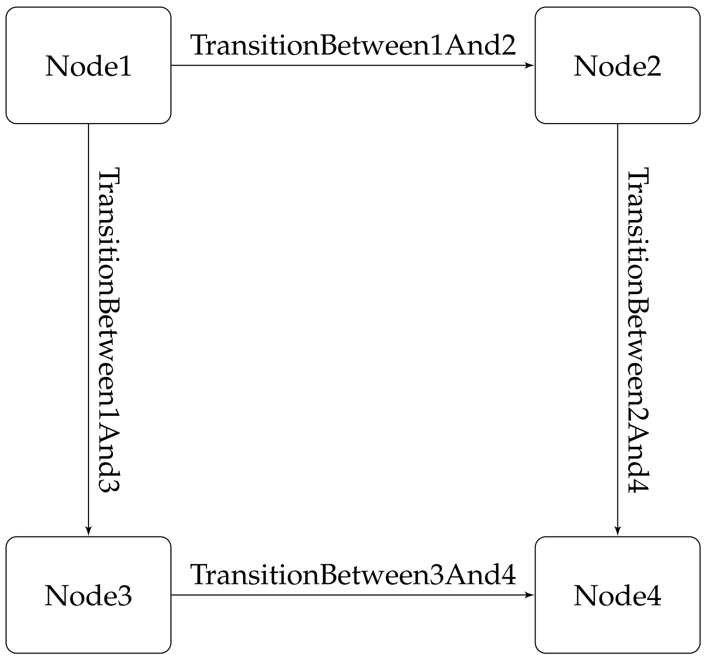


Figure 4.2: Example of graph described through the Trivial Graph Format

effectively move in the environment are contained in another structure, the *Pathfinding Graph*, which is represented similarly to the finite state machine in the definition but has different purpose and usage, that will be described later

The rules for encoding a Finite State Machine, as used in our project, into a TGF file are the following:

- The Finite State Machine associated with each NPC must be described in a file named `<NameOfTheNPC>FSM.tgf`, placed inside the *FiniteStateMachines* folder;

- The initial state for the Finite State Machine must be described by the first line of the file;
- The syntax for describing the states is:
  `NodeID NodeLabel`
  where `NodeID` is an identification number for the state and `NodeLabel` is a string containing the full name of the chosen position in the game environment (e.g., `NodeA`);
- The syntax for describing the transitions is:
  `SourceNodeID DestinationNodeID EdgeLabel`
  where `SourceNodeID` and `DestinationNodeID` are the IDs of the two states connected by the transition, and `EdgeLabel` is a string without spaces containing the name of the transition (e.g., `MoveTo`);
- The declaration of the states and the declaration of the transitions must be divided by a line whose content is the single character `#`.

Field summary:

- CurrentState – Contains the current state of the finite state machine
- NextActions – Contains the transitions available from the current state
- ReadNodes – Contains a the states that compose the finite state machine

Method summary:

- SaveAsXML() – Serializes the finite state machine to a XML file
- Start() – Creates the finite state machine, parsing the correspondent TGF file

The Finite State Machine also encloses a method called `SaveAsXML()`, which essentially *serializes* the Finite State Machine kept in memory to a XML file; this function has been extremely useful for the integration of the C# scripts developed in Unity with the servers offering the Composition service (at first the SM4LL Web Service, and later the JaCO Web Service), that accepted XML as the exchange format for the inputs and the output. For example, figure 4.3 shows the XML serialization of the finite state machine of figure 4.2.

```xml
<?xml version="1.0" encoding="utf-8"?>
<service xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:
    schemaLocation="http://www.sm4all-project.eu/composition/sbl ../sbl.xsd"
    name="Example" stid="http://www.sm4all-project.eu/composition/sample/
    Example" class="service-type">
  <ts>
    <state name="Node1" type="initial-final">
      <transition action="TransitionBetween1And2">
        <target state="Node2" />
      </transition>
      <transition action="TransitionBetween1And3">
        <target state="Node3" />
      </transition>
    </state>
    <state name="Node2" type="final">
      <transition action="TransitionBetween2And4">
        <target state="Node4" />
      </transition>
    </state>
    <state name="Node3" type="final">
      <transition action="TransitionBetween3And4">
        <target state="Node4" />
      </transition>
    </state>
    <state name="Node1" type="final" />
  </ts>
</service>
```

Figure 4.3: XML serialization of the finite state machine of figure 4.2

4.2 PathfindingGraph

The *Pathfinding Graph* is an auxiliary data structure that we developed as an extension of the various types of graph provided by the A* Pathfinding Project. The main motivation that led us to create a new class instead of reusing the given ones was to be able to maintain the locations of the environment we are interested in as `GameObjects`, to be easily accessible and modifiable within the Unity Editor. Similar to the Finite State Machine discussed earlier, also the *Pathfinding Graph* takes advantage of the Trivial

Graph Format encoding, that allows the graph to be editable by the user by changing the arrangement of nodes and connections in the correlated file and built at runtime when the Unity executable is launched.
The guidelines for describing a *Pathfinding Graph* through the TGF format are:

- The *Pathfinding Graph* associated to a specific NPC must be described in a file named `<NameOfTheNPC>Graph.tgf`, placed inside the *PathfindingGraphs* folder;
- The initial *Path Node* of the *Pathfinding Graph* must be described by the first line of the file;
- The syntax for describing the *Path Nodes* is:
  `NodeID NodeLabel`
  where `NodeID` is an identification number for the *Path Node* and `NodeLabel` is a string containing the full name of the chosen *Path Node* (e.g., `NodeA`);
- The syntax for describing the *Paths* is:
  `SourceNodeID DestinationNodeID`
  where `SourceNodeID` and `DestinationNodeID` are the IDs of the two *Path Nodes* connected by the *Path*;
- The declaration of the *Path Nodes* and the declaration of the *Paths* must be divided by a line whose content is the single character `#`.

Once the *Pathfinding Graph* is in memory, the A* Pathfinding Project scripts will take care of the rest: as we will introduce later in the section regarding the `MoveNPC.cs` script, to make the robot move from its current position to the location the user chose clicking one of the buttons of the GUI, all we have to do is to set the desired location as the new target point, and then ask A* to search for a path for reaching it; the A* script will use the information contained in the *Pathfinding Graph* to find a possible way to get there and, if there is one, it will instruct the robot step-by-step until it fulfills the given choice.

Method summary:

- GetGraphNode() – Returns the node of the graph correspondent to the GameObject given as input
- GetPathNodes() – Returns the list of Transform objects correspondent to the Path Nodes
- ParseFile() – Parses the TGF file and builds the nodes and connections used by the Scan() function
- Scan() – Creates the Pathfinding Graph to be used by A* Pathfinding Project
- SetNodeAsWalkable() – Sets the Path Node given as input as walkable, so to be considered when looking for path

4.3 TargetBehavior

The *Target Behavior* is a sub-class of the `FiniteStateMachine`, and shares with this data structure the majority of its methods and properties. The main reason behind the choice to realize such a sub-class is due to some slightly different rules used to encode this object into a Trivial Graph Format file: while in a Finite State Machine it is sufficient to describe just the name of the desired action in the `EdgeLabel`, as shown in the following example:

```
1 NodeA
2 NodeB
3 NodeC
#
1 2 MoveTo
2 3 MoveTo
1 3 MoveTo
```

the *Target Behaviour* requires the user to specify the full name of the selected action, composed by the action name used in the finite state machine and the destination node, as shown in this second example:

```
1 A
2 B
3 C
#
1 2 MoveToNodeA
2 3 MoveToNodeB
1 3 MoveToNodeB
```

Please note that, in this case, the definition of the nodes refers to the states that compose the *Target Behavior* transition system, and hence do not refer to the various *Path Nodes* available in the game environment, that are used by the non-player characters (available behaviors). Another guideline that is different between the finite state machine and the *Target Behaviour* is that the latter is required to be expresses in a file named `TargetBehavior.tgf`.

Field summary:

- CurrentState – derived from the FiniteStateMachine script
- NextActions – derived from the FiniteStateMachine script
- ReadNodes – derived from the FiniteStateMachine script

Method summary:

- SaveAsXML() – Serializes the Target Behavior to an XML file
- Start() – Creates the target behavior, parsing the correspondent TGF file

4.4 ActionLookupTable & ActionLookupTableGenerator

The Action Lookup Table contains what, in Section 2.5.2, we called the transition relation for the *enacted system behavior*: essentially it is a big table where we store as entries the possible states in which the entire system (composed by the available behaviors) might be, the actions that could be performed in a particular state, and what their outcome could be, i.e. what the system situation will look like after executing a specific action starting from a particular state.

In the first iteration of the Angry Bots patrolling domain, the Action Lookup Table was built procedurally by the `ActionLookupTableGenerator.cs` script, based on the finite state machines related to each non-player character present in the Unity scene. In particular, we generate the Cartesian product among all the behaviors' possible states and all the possible transitions from one state of the system to another. It turned out that the Action Lookup Table built following this approach was quite large even in the case of 3 robots (about 5000 possible configurations), already suffering an exponential growth that becomes infeasible to handle with the addition of new non-player characters.

In order to make feasible the search and the retrieval of the current state in such a big table, we implemented it as a *hash table*, which is a particular data structure that associates *keys* to *values* in such a way that the average cost for each lookup in this kind of storage is independent of the number of elements maintained. To make this possible, we created an object called `PossibleState` to hold one possible configuration of the system, and we made it implement the `IEquatable<T>` and the `IEqualityComparer<T>` C# interfaces, necessary to give to the object the possibility of being considered as a key in the hash table, and thus retrieved by means of its hash code. This results in a slight delay in the startup phase, when the script cycles over all the available behaviors to obtain all of their possible states and combines them in this data structure, and an excellent retrieval time during run-time.

In the second iteration we aimed at using the same data structure to handle the result of a behavior composition engin,e such as the implementation of the Composition Engine from the SM4LL Project (described in Section 5.2. We modified the `ActionLookupTable.cs` script to also be able to parse the XML file returned as output by the Web Service and build the Action Lookup Table at runtime, similar to what the `FiniteStateMachine.cs` and the `PathfindingGraph.cs` scripts do with the files encoded with the Trivial Graph Format.

The Composition Engine will not return the complete table of system states, available next actions, and resulting successor states, but only the states needed to map the desired target behavior to the subset of behaviors' states and transitions required to realize it, thoroughly ignoring the unnecessary ones. Obviously, this results in a much smaller lookup table, with immediate benefits in both the space needed to maintain all this information in memory, and especially the startup time required to build the auxiliary data structure employed at runtime.

ActionLookupTable
Method summary:

- GetNextActions() – Returns the list of actions that can be executed from the current state
- ReadFromJson() – Creates the Action Lookup Table parsing a JSON file
- ReadFromXML() – Creates the Action Lookup Table parsing a XML file

ActionLookupTableGeneration
Method summary:

- BuildTable() – Generates the complete enacted system behavior and serializes it to a JSON file

4.5 ControllerGUI

The `ControllerGUI` script manages two of the most important concepts we wanted to experiment with in this project: the *controller* and the interaction with the user.

The Graphical User Interface that allow the player to instruct the various non-player characters has been implemented with UnityGUI, a collection of scripts developed directly by the programmers at Unity Technologies and integrated into the Unity Editor. The way of employing it into the game loop slightly differs from the standard method: usually, the code needed to get the input from the user and update accordingly the game world for one frame is contained into the `Update()` function; in the case of GUI programming, Unity requires the code to be encapsulated in a different method, called `OnGUI()`.

Another atypical feature of this package of scripts is that, in fact, the method that the programmer calls to draw a graphic control on the screen also return a boolean value which tells whether the player interacted with it, e.g. whether a button has been clicked or not. This implementation choice leads to the writing of event-management code that at a first glance might seem counter-intuitive, but after a closer look it is easy to see that it is simpler and natural, as the following example code shows:

```
function OnGUI () {
  if( GUI.Button(Rect(10, 10, 100, 200), "Click Me!") ) {
    print("User has clicked the button!");
  }
}
```

The options that are displayed to the user are taken from the next actions available from the current system state, and specifically only among those the Composition Engine considered as essential for realizing the user-defined target behavior. As we have already said, our goal was to put the player in charge of being the *system controller*, according to the formal definition given in Chapter 2.5, so that he is able to ask the system to execute one specific action and, by looking up at the Action Lookup Table discussed earlier, observe one of the available behavior fulfill his order while still having the guarantee that this would not lead the system to a non-valid state, in which he might receive a *failure* status code upon his new request (actually, according to the formal definition, the controller would designate the *undefined* behavior to execute the desired operation).

When the user interacts with the GUI by pressing one of the buttons, the `ControllerGUI` will search in the Action Lookup Table which non-player character is capable of dealing with the player's request, and subsequently will retrieve from it the script that manages to act the required behavior (e.g., `MoveNPC` in the case of action `MoveTo`) and invoke it. During the execution of an action, the Graphical User Interface is temporarily substituted by a message reporting that an action is being executed, kindly asking the user to wait the completion of this process to select another option. A screenshot of the Controller GUI is shown in figure 4.4.

Field summary:

- actionLookupTable – Table of <PossibleState, NextActions> entries
- ExecutingAction – Boolean value that expresses whether an action is being executed
- npcsList – List of the NPCs present in the Scene
- nodesList – List of all the points of interest

Method summary:

- GenerateFiles() – Writes all the files necessary for the interaction with the SM4LL service
- OnGUI() – Called by Unity, displays on the screen the GUI for selecting the next action
- Start() – Retrieves the npcsList, the nodesList and the actionLookupTable

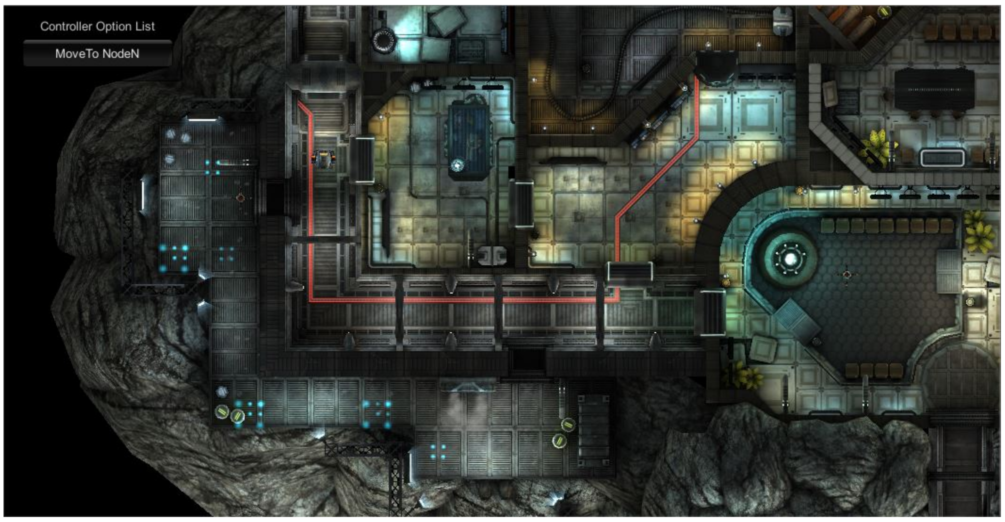


Figure 4.4: A screenshot of the Controller GUI

4.6 MoveNPC

The `MoveNPC` script handles the code to be executed upon the request of a `MoveTo` action which, since the example is based on patrolling and the way of achieving this goal is to send the robots to control specific locations in the environment, we believe is the most important one. This script, obviously, heavily relies on the facilities offered by the A* Pathfinding Project library. When, by clicking on one of the buttons present in the Graphical User Interface, the user expresses the intention to send a robot to a point of interest of the Angry Bots level, the controller invites the appropriate non-player character for the job by calling the `SetTargetPosition()` function. Upon its invocation, the method controls if, effectively, the requested location belongs to the ones assigned to the current NPC and, if this is the case, asks the A* script to find a path that connects the current robot position to the wished target, taking into account only the `Composition Graph` associated with the activated behavior. Once, and if, a valid path is found, the `MoveNPC` script starts to guide the relocation of the chosen non-player character in a step-by-step manner: it will instruct the robot on how to reach the first waypoint of the route; then, when it is within a pre-defined distance (called in the script `goalReachedDistance`), it will start directing it to the next spot, and so on. The script is also responsible of playing the correct animation when the NPCs move or remain still in one location, and also opens up in the user's screen a personalized camera when one of the non-player characters is moving, so that the player can understand immediately which is the robot the controller delegated for the execution of the desired action.

Method summary:

- FixedUpdate() – Called by Unity, calculates the movement of the NPC for the next frames
- OnPathComplete() – Called by A* Pathfinding Project, sets the path just found as the one to be followed by the NPC
- Start() – Generates the Pathfinding Graph of the NPC and attaches the Seeker component for A* Pathfinding

CHAPTER 5

NPC BEHAVIOR COMPOSITION USING SM4LL

5.1 The Roman model

Services are software artifacts, possibly distributed and built on top of different technologies, that export a description of themselves, are accessible to external clients and communicate through a commonly known, standard interface which enables interoperability. More in general, Service Oriented Computing (SOC) is a computing paradigm whose basic elements are services, that can be used as building blocks to devise other services. A classical example of such a paradigm is provided by *Web services*, i.e., applications published over the Internet and self-described, usually built by different companies and relying on different technologies, which share a same communication protocol, namely SOAP. No constraints are required over the internal structure of each Web service, but they are all required to be published, compliant with the same communication protocol and to export a description of their interface, so as to facilitate access and communication.

The *automated service composition* is the problem of automatically combining a set of available services, so as to meet a desired specification. This particular problem has been tackled with different techniques, starting by exploiting a reduction to satisfiability in a well-known logic of programs, namely the Propositional Dynamic Logic [Berardi, Calvanese, De Giacomo, Lenzerini, et al., 2003; Berardi, Calvanese, De Giacomo, and Mecella, 2005]. More recently, another approach has been proposed based on computing compositions by exploiting (variants of) the formal notion of simulation [De Giacomo, Patrizi, and Sardiña, 2007]. Interestingly, through this, the case where the state of services is only partially observable has been also addressed. The solution technique directly appeals to techniques for Linear Time Logic (LTL) synthesis, to model-check a game structure representing a so-called safety-game. Since this can be realized in practice on top of symbolic model checking technologies, the approach gained a high level of scalability.

In the literature, several approaches to service modeling have been proposed. A WSDL (Web Services Description Language) description exports a *functional* specification of a service, that is, from an abstract standpoint, the set of operations provided by the service, along with the corresponding format of messages exchanged. Unfortunately, this input-output approach does not allow for *conversation specification*, i.e., for putting constraints on the order that operations should be executed in.

In the Roman model, originally introduced in [Berardi, Calvanese, De Giacomo, Lenzerini, et al., 2003], services export their *behavioral* features by means of a language that represents transition systems, i.e., Kripke structures whose transitions are labeled by service's operations, under the assumption that each legal run of the system corresponds to a conversation supported by the service. A first advantage brought by such a model is its *generality* with respect to service integration, in the sense that it is abstract enough to serve as conceptual model for several classes of scenarios. Second, from the SOC viewpoint, it provides a behavioral, stateful, service representation, which allows for describing those inter-operation constraints that current languages, e.g., WSDL, do not capture.

The approach to service composition described in the Roman model falls into the client-tailored service class, in which the community ontology (the agreed upon reference semantics between the services) is built mainly taking into account the client independently from the services available, and its distinguishing features can be summarized as follows:

- The available services are grouped together into a so-called *community*;
- Services in a community share a common set of actions Σ, the actions of the community;
- Actions in Σ denote (possibly complex) interactions between service and clients. As a result of an interaction the client may acquire new information (not necessarily modeled in the description) that may affect the next interaction.
- The behavior of each available service is described in terms of a finite transition system that uses only actions from Σ.
- The desired service, called the target service, is itself described as a finite, deterministic transition system that uses actions from Σ. Determinism here captures the absence of uncertainty over the desired behavior.
- The orchestrator has the ability of scheduling services on a *step-by-step* basis.

In this approach, the composition synthesis task consists in synthesizing an orchestrator able to coordinate the community services so as to mimic the behavior of the target service. Differently put, the behavior obtained by coordinating the services should present no differences, from the client perspective, with the target service.

5.2 Target Behavior and Composition as a Web Service

To obtain the composition needed to make the user control the execution of the target behavior, we employed the OffSEn (Off-line Synthesis Engine) developed for the Smart Homes For All (SM4LL) Project [1]. This is a Composition Engine written in Java

[1]`http://sm4ll-project.eu`

and deployed in a Ubuntu 10.04 "Lucid Lynx" virtual machine, codenamed RomanCE; the interaction with the system is made possible through a web interface, developed with JavaServer Pages and servlets, and deployed within the Apache Tomcat servlet container.

The engine accepts as input a collection of XML files, each encoded following the rules defined in the XSD (XML Schema Definition) correlated to each component of the behavior composition system: the *Service Behavior Language*, or *SBL*, for the available and the target behaviors, and the *Service Deployment Descriptor*, or *SDD*, for the XML file listing the available behaviors (o services, as they are called within the SM4LL enviroment). An example of an XML file compliant with the *Service Behavior Language* schema is the following:

```xml
<?xml version="1.0" encoding="utf-8"?>
<service xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:
    schemaLocation="http://www.sm4all-project.eu/composition/sbl ../sbl.xsd"
    name="MyEnemyMech" stid="http://www.sm4all-project.eu/composition/sample/
    MyEnemyMech" class="service-type">
  <ts>
    <state name="NodeA" type="initial-final">
      <transition action="MoveToNodeN">
        <target state="NodeN" />
      </transition>
      <transition action="MoveToNodeG">
        <target state="NodeG" />
      </transition>
      <transition action="TakeSnapshotNodeA">
        <target state="NodeA" />
      </transition>
    </state>
    <state name="NodeB" type="final">
      <transition action="MoveToNodeN">
        <target state="NodeN" />
      </transition>
      <transition action="MoveToNodeM">
        <target state="NodeM" />
      </transition>
      <transition action="TakeSnapshotNodeB">
        <target state="NodeB" />
      </transition>
    </state>
    <state name="NodeC" type="final">
      <transition action="MoveToNodeM">
        <target state="NodeM" />
      </transition>
      <transition action="MoveToNodeD">
        <target state="NodeD" />
      </transition>
      <transition action="TakeSnapshotNodeC">
        <target state="NodeC" />
      </transition>
    </state>
    <!-- Code omitted for brevity -->
  </ts>
</service>
```

The implementation choice to store the target of a given transition as a child node is due to the fact that, in general, the available behaviors are *partially controllable*, and hence we formalize them using *non-deterministic* transition systems. In this kind of structure, a transition could bring the system into different states if triggered at different times; it is very simple to render this concept inside the XML Schema with the usage of children nodes, while it would have been rather tricky to manage this situation using node attributes, especially in the *unmarshalling* phase, i.e. when converting the XML file to an object maintained in memory.

The output of the application, the *composition*, is encapsulated within an XML file encoded with another XSD, called *Composition Behaviour Language* (*CBL*). An example of composition described through this schema is the following:

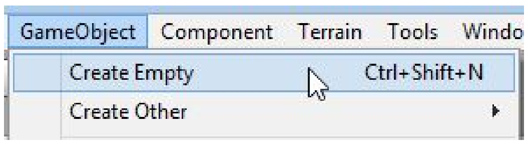
```
<?xml version="1.0" encoding="UTF-8"?>
<composition xmlns="http://www.sm4all-project.eu/composition/cbl" xmlns:xsi="
    http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www
    .sm4all-project.eu/composition/cbl http://www.dis.uniroma1.it/~cdc/sm4all
    /proposals/servicemodel/latest/src/compositions/cbl.xsd" name="
    TargetBehaviour">
  <conversational-target-state name="a" type="initial-final">
    <conversational-orchestration-state>
      <community-state>
        <service-conversational-state service="MyEnemyMineBot" state="NodeH"/
            >
        <service-conversational-state service="MyEnemyMech" state="NodeA"/>
        <service-conversational-state service="MyEnemyMineBot1" state="NodeT"
            />
      </community-state>
      <transition action="MoveToNodeN" invoke="MyEnemyMech">
        <target state="b"/>
      </transition>
    </conversational-orchestration-state>
  </conversational-target-state>
  <conversational-target-state name="b" type="final">
    <conversational-orchestration-state>
      <community-state>
        <service-conversational-state service="MyEnemyMineBot" state="NodeH"/
            >
        <service-conversational-state service="MyEnemyMech" state="NodeN"/>
        <service-conversational-state service="MyEnemyMineBot1" state="NodeT"
            />
      </community-state>
      <transition action="MoveToNodeP" invoke="MyEnemyMineBot1">
        <target state="c"/>
      </transition>
    </conversational-orchestration-state>
  </conversational-target-state>
  <conversational-target-state name="c" type="final">
    <conversational-orchestration-state>
      <community-state>
        <service-conversational-state service="MyEnemyMineBot" state="NodeH"/
            >
        <service-conversational-state service="MyEnemyMech" state="NodeN"/>
        <service-conversational-state service="MyEnemyMineBot1" state="NodeP"
            />
      </community-state>
      <transition action="TakeSnapshotNodeP" invoke="MyEnemyMineBot1">
        <target state="d"/>
      </transition>
    </conversational-orchestration-state>
  </conversational-target-state>
  <!-- Code omitted for brevity -->
</composition>
```

5.3 Getting started with the SM4LL Web Service

5.3.1 How To Create Prefabs

A `Prefab` in Unity is a prototype of any kind of `GameObject` you could need: you can have `Prefabs` for models, cameras, lights, audio sources, and so on. With `Prefabs`, you concentrate your effort on creating one "template" `GameObject`, with all the characteristics and the scripts you need, and declaring it as a `Prefab`; afterwards, you can deploy exact replicas of that `GameObject` simply by instantiating the `Prefab` you have created. For this project we provide two `Prefabs`, called `MyEnemyMech` and `MyEnemyMineBot`, that you can use to extend the project; if you want to create your own `Prefab`, you should:

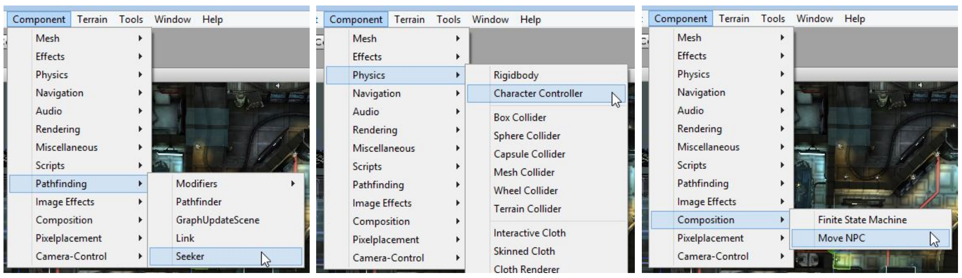
1. Create a new `GameObject` in the Hierachy Panel (GameObject → Create Empty in the Menu Bar);



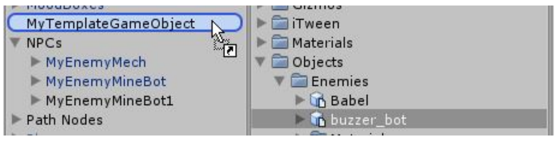
2. Attach to the `GameObject` you have just created the following scripts:

   - `Seeker`: from the Menu Bar select Component → Pathfinding → Seeker;
   - `Character Controller`: from the Menu Bar select Component → Physics → CharacterController;

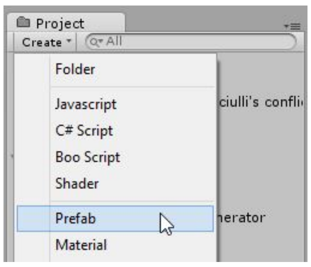
- `MoveNPC`: from the Menu Bar select Component → Composition → Move NPC;
- `Finite State Machine`: from the Menu Bar select Component → Composition → Finite State Machine;



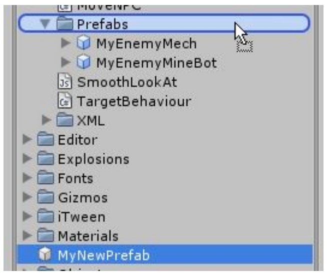
3. Drag the 3D model you chose for the `Prefab` onto the `GameObject` you have created (in this example, I took a model already present in the Angry Bots demo, but you can use any 3D model you like);



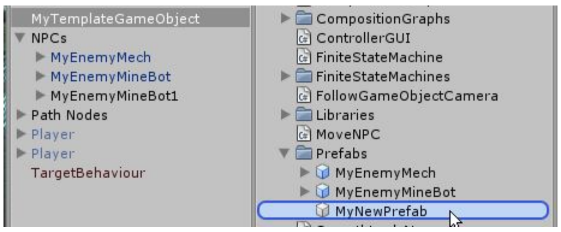
4. In the Hierachy Panel, create a new `Prefab` by selecting `Create`, and then `Prefab`;



5. Rename the `Prefab` with any name you want, and drag it into the *Prefabs* folder;

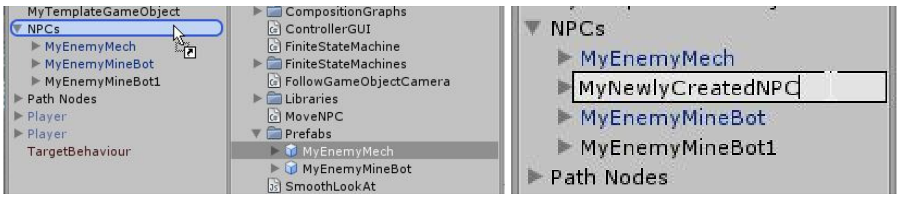


6. Drag the "template" `GameObject` you have created onto the `Prefab`.

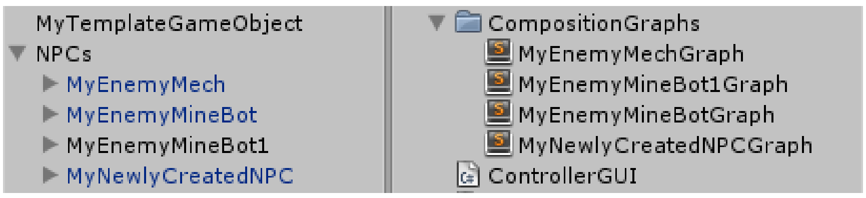


5.3.2 How to add a new NPC

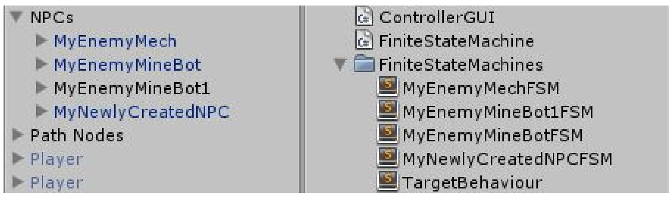
1. Drag the `Prefab` into the Hierarchy Panel as a child of the *NPCs* `GameObject`, and rename the newly created `GameObject` with any name you like;



2. Create a new `.tgf` file in the *PathfindingGraphs* folder, and fill it with the *PathfindingGraph* you want to describe for the new NPC, following the guidelines exposed in Section 4.2;



3. Create a new `.tgf` file in the *FiniteStateMachine* folder, and fill it with the Finite State Machine you want to create for the new NPC, following the guidelines exposed in Section 4.1;

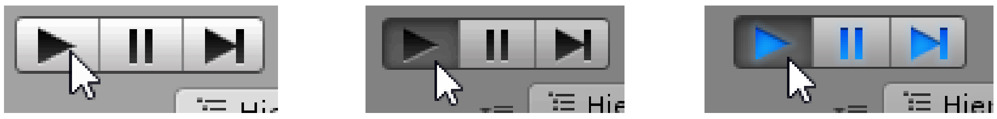


4. If necessary, modify the `TargetBehaviour.tgf` file to include actions performed by the new NPC.

5.3.3 How to make the Composition Engine work

The procedure described below must be followed only if you modified one of the Finite State Machines or the *PathfindingGraphs*, and hence the Composition may need to be modified as well. If there are no modifications, you can simply press the Play button in the Unity interface to make the computed Composition work.

1. In the Unity interface, press once the Play button (figure 5.1a), wait for the completion of the startup phase (figure 5.1b) and, when the project fully starts (figure 5.1c), press the Play button once again. During the startup phase, the project will load in memory the Finite State Machines and the *PathfindingGraphs* associated with each *NPC* and will create the `.zip` files we need to give as input to the Composition Engine;



(a) Idle **(b)** Startup phase **(c)** Started

Figure 5.1: Various states of the Play button

2. Start the RomanCE Virtual Machine, using `romance` as password, and follow the instructions contained in the `readme` file on the Desktop of the Ubuntu Operating System to start the Composition Engine (figure 5.2);

3. Set the `XML` folder as a *Shared Folder* between your host and guest Operating Systems and mount it (this step varies depending on the virtualization software you are using; for Oracle VM VirtualBox follow this link: HOWTO: Use Shared Folders)

4. In the Services tab of the OffSEn web interface (figure 5.3):

   - Click **Deploy a Service Archive**;
   - Click **Browse...**
   - In the **File Upload** dialog, search for the *Shared Folder*, select the `services.zip` file and click **Open**;
   - Back in the Service tab, click **Deploy**.

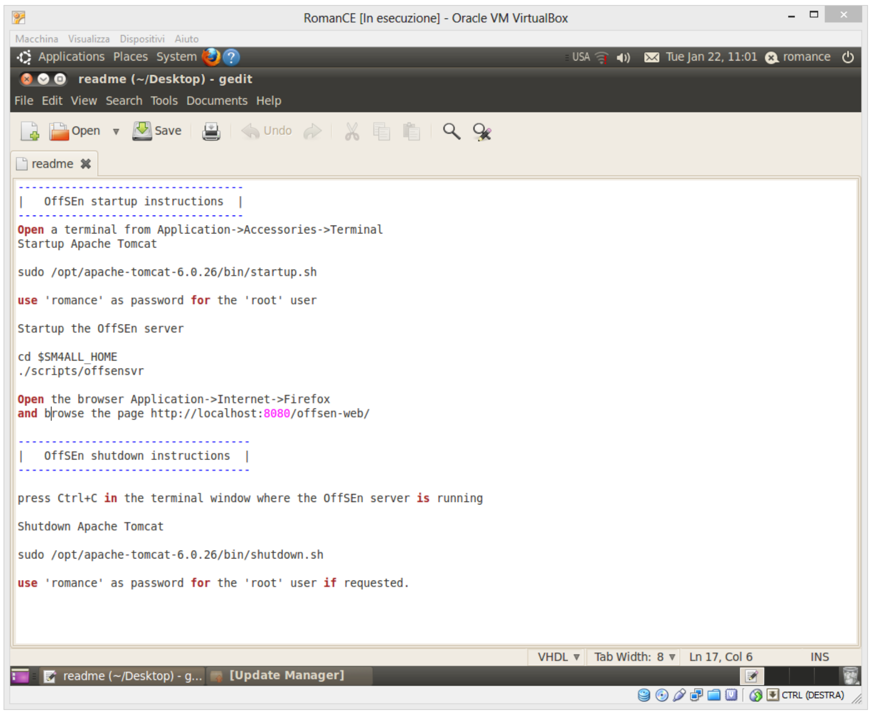
5. In the Targets tab of the OffSEn web interface figure 5.4):

   - Click **Deploy a new Target**;
   - Click **Browse...**
   - In the **File Upload** dialog, search for the *Shared Folder*, select the `target.zip` file and click **Open**;
   - Back in the Targets tab, click **Deploy**.

6. Still in the Targets tab, click **Start Synthesis** (figure 5.5);

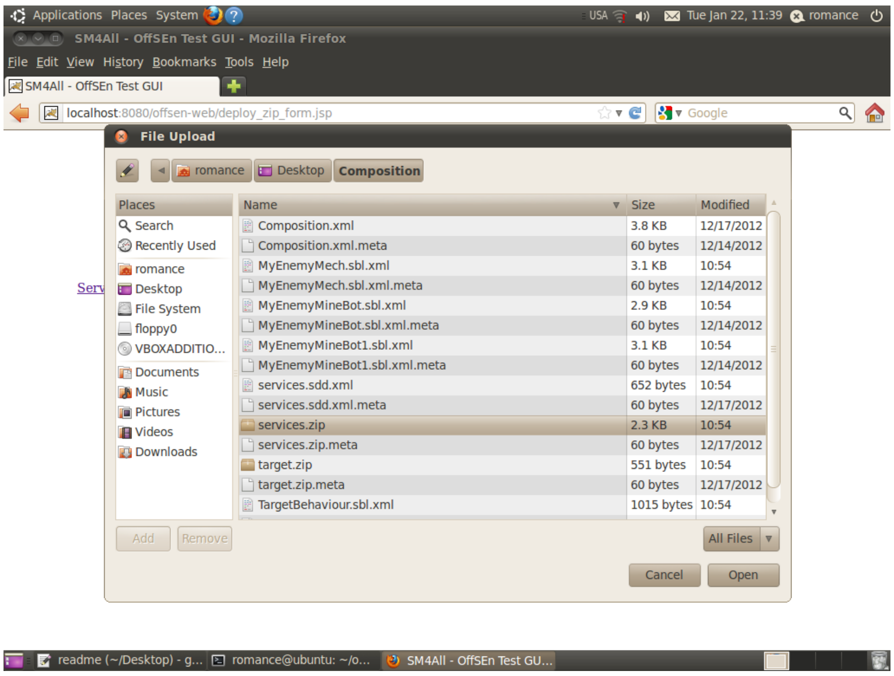
7. In the Compositions tab, right-click **View**, select **Save Link As...**, rename the file to *Composition.xml* and save it in your *Shared Folder* (figure 5.6);

8. Shut down the RomanCE Virtual Machine and, back to Unity, press the Play button like in Step 1 (obviously, this time don't press it again until you want to stop exploring the project).


RomanCE [In esecuzione] - Oracle VM VirtualBox
Macchina Visualizza Dispositivi Aiuto
Applications Places System
USA Tue Jan 22, 11:01 romance
readme (~/Desktop) - gedit
File Edit View Search Tools Documents Help
Open Save Undo
readme
VHDL Tab Width: 8 Ln 17, Col 6 INS
readme (~/Desktop) - g... [Update Manager]
CTRL (DESTRA)

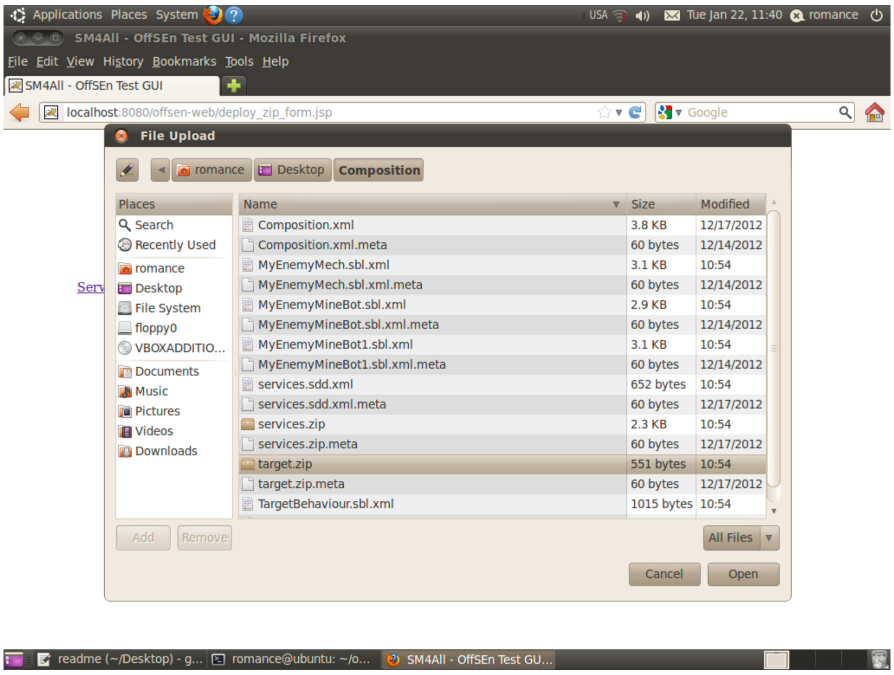
```
----------------------------------
|   OffSEn startup instructions  |
----------------------------------
Open a terminal from Application->Accessories->Terminal
Startup Apache Tomcat

sudo /opt/apache-tomcat-6.0.26/bin/startup.sh

use 'romance' as password for the 'root' user

Startup the OffSEn server

cd $SM4ALL_HOME
./scripts/offsensvr

Open the browser Application->Internet->Firefox
and browse the page http://localhost:8080/offsen-web/

-----------------------------------
|   OffSEn shutdown instructions  |
-----------------------------------

press Ctrl+C in the terminal window where the OffSEn server is running

Shutdown Apache Tomcat

sudo /opt/apache-tomcat-6.0.26/bin/shutdown.sh

use 'romance' as password for the 'root' user if requested.
```

Figure 5.2: The RomanCE Virtual Machine in Oracle VirtualBox


Applications Places System
USA Tue Jan 22, 11:39 romance
SM4All - OffSEn Test GUI - Mozilla Firefox
File Edit View History Bookmarks Tools Help
SM4All - OffSEn Test GUI
localhost:8080/offsen-web/deploy_zip_form.jsp
Google
File Upload
romance Desktop Composition
Places
Search
Recently Used
romance
Desktop
File System
floppy0
VBOXADDITIO...
Documents
Music
Pictures
Videos
Downloads
Add Remove
All Files
Cancel Open
readme (~/Desktop) - g... romance@ubuntu: ~/o... SM4All - OffSEn Test GU...

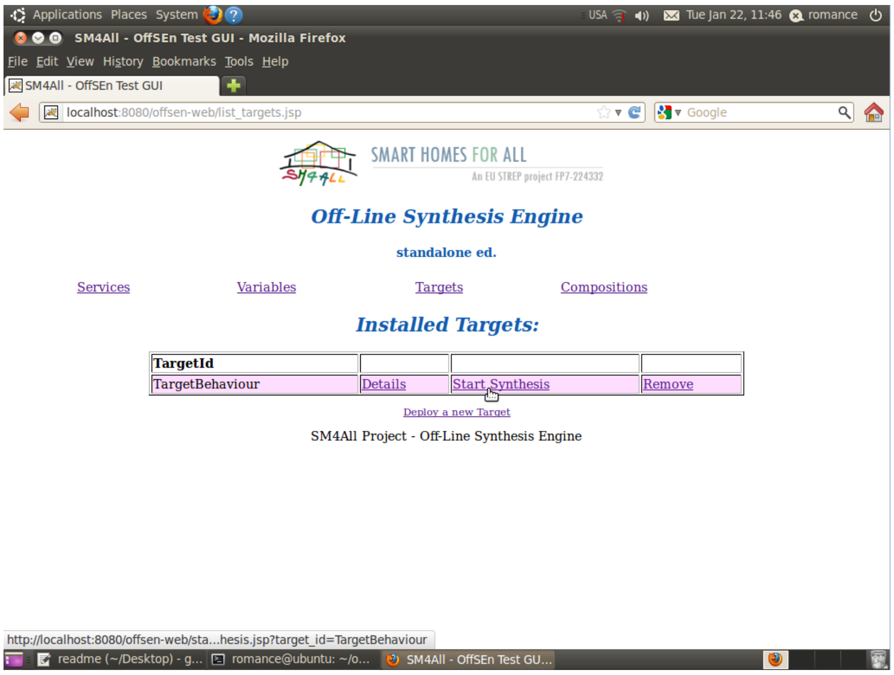
Name	Size	Modified
Composition.xml	3.8 KB	12/17/2012
Composition.xml.meta	60 bytes	12/14/2012
MyEnemyMech.sbl.xml	3.1 KB	10:54
MyEnemyMech.sbl.xml.meta	60 bytes	12/14/2012
MyEnemyMineBot.sbl.xml	2.9 KB	10:54
MyEnemyMineBot.sbl.xml.meta	60 bytes	12/14/2012
MyEnemyMineBot1.sbl.xml	3.1 KB	10:54
MyEnemyMineBot1.sbl.xml.meta	60 bytes	12/14/2012
services.sdd.xml	652 bytes	10:54
services.sdd.xml.meta	60 bytes	12/17/2012
services.zip	2.3 KB	10:54
services.zip.meta	60 bytes	12/17/2012
target.zip	551 bytes	10:54
target.zip.meta	60 bytes	12/17/2012
TargetBehaviour.sbl.xml	1015 bytes	10:54

Figure 5.3: Uploading the Services


Applications Places System
USA Tue Jan 22, 11:40 romance
SM4All - OffSEn Test GUI - Mozilla Firefox
File Edit View History Bookmarks Tools Help
SM4All - OffSEn Test GUI
localhost:8080/offsen-web/deploy_zip_form.jsp
Google
File Upload
romance Desktop Composition
Places
Search
Recently Used
romance
Desktop
File System
floppy0
VBOXADDITIO...
Documents
Music
Pictures
Videos
Downloads
Serv
Add Remove
All Files
Cancel Open
readme (~/Desktop) - g...
romance@ubuntu: ~/o...
SM4All - OffSEn Test GU...

Name	Size	Modified
Composition.xml	3.8 KB	12/17/2012
Composition.xml.meta	60 bytes	12/14/2012
MyEnemyMech.sbl.xml	3.1 KB	10:54
MyEnemyMech.sbl.xml.meta	60 bytes	12/14/2012
MyEnemyMineBot.sbl.xml	2.9 KB	10:54
MyEnemyMineBot.sbl.xml.meta	60 bytes	12/14/2012
MyEnemyMineBot1.sbl.xml	3.1 KB	10:54
MyEnemyMineBot1.sbl.xml.meta	60 bytes	12/14/2012
services.sdd.xml	652 bytes	10:54
services.sdd.xml.meta	60 bytes	12/17/2012
services.zip	2.3 KB	10:54
services.zip.meta	60 bytes	12/17/2012
target.zip	551 bytes	10:54
target.zip.meta	60 bytes	12/17/2012
TargetBehaviour.sbl.xml	1015 bytes	10:54

Figure 5.4: Uploading the Target Behavior


Applications Places System
USA Tue Jan 22, 11:46 romance
SM4All - OffSEn Test GUI - Mozilla Firefox
File Edit View History Bookmarks Tools Help
SM4All - OffSEn Test GUI
localhost:8080/offsen-web/list_targets.jsp
Google
SMART HOMES FOR ALL
An EU STREP project FP7-224332
Services
Variables
Targets
Compositions
Deploy a new Target
http://localhost:8080/offsen-web/sta...hesis.jsp?target_id=TargetBehaviour
readme (~/Desktop) - g...
romance@ubuntu: ~/o...
SM4All - OffSEn Test GU...

Off-Line Synthesis Engine

standalone ed.

Installed Targets:

TargetId			
TargetBehaviour	Details	Start Synthesis	Remove

SM4All Project - Off-Line Synthesis Engine

Figure 5.5: The Start Synthesis hyperling

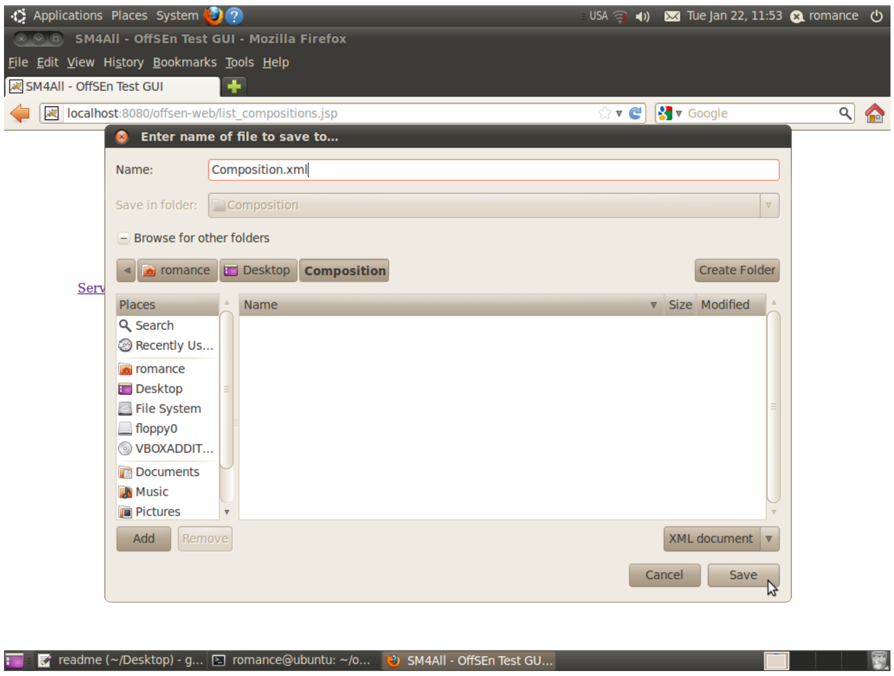

Applications Places System
SM4All - OffSEn Test GUI - Mozilla Firefox
File Edit View History Bookmarks Tools Help
localhost:8080/offsen-web/list_compositions.jsp
Enter name of file to save to...
Name: Composition.xml
Save in folder: Composition
Browse for other folders
romance Desktop Composition
Create Folder
Places
Name
Size
Modified
Add
Remove
XML document
Cancel
Save

Figure 5.6: Saving the composition

5.4 Results with the SM4LL Web Service

The SM4LL Off-line Synthesis Engine immediately resulted to be suitable for our needs: once we wrote the classes capable of representing the needed inputs in the XML Schema Definitions accepted by the OffSEn and to translate the output file from XML to the kind of information maintained within the *Action Lookup Table*, we managed to easily visualize the behavior composition concept using the Unity framework we worked on.

The only disadvantage of this Composition Engine is that it requires a big effort from the user's point of view: each time one of the available behaviors or the target behavior gets modified, to obtain a new valid composition for the system the user would need to locate the ZIP archives containing the various XML files, connect to the web interface provided by the engine, upload the archives one at a time, wait for the composition to be generated, and be careful to put the computed output in the right folder. In the situation in which the web interface is not deployed to be accessible from outside the virtual machine, the user will also be required to launch the RomanCE environment, find a way to share folders and files between the host and the guest Operating Systems, and launch the Apache Tomcat servlet container.

This intricate way of interfacing with the system motivated us to develop a new Web Service, compliant with the REST principles and fully accessible via the Internet, such that it can be accessed not only from the Unity framework but from every piece of code capable of sending and receiving HTTP messages. The result of our work goes under the name of *JaCO* (Java-based Composition-Oriented Web Service), and it will be described extensively in the next chapter.

CHAPTER 6

BEHAVIOR COMPOSITION AS A RESTFUL WEB SERVICE

6.1 JaCO: a Java server for behavior composition

As stated in the last section of the previous chapter, we decided to create a Web Service, written in Java and compliant with the REST principles exposed above, whose interaction was fully driven by the sending and the receiving of HTTP messages, so that it was simple to make the composition request completely transparent and effortless to the user, but also powerful enough to permit interested users and other programs to access the services offered by the server via the Web. The name JaCO stands for Java-based Composition-Oriented Web Service.

The basis for the development of this project is *Jersey* [1], which is Oracle's reference implementation for building RESTful Web Services following the guidelines described in the Java Specification Request (JSR) 311, "JAX-RS: Java API for RESTful Web Services". An important reference kept into account during the design and the realization of this

[1] `http://jersey.java.net/`

application is the tutorial "REST with Java (JAX-RS) using Jersey" written by Lars Vogel [Vogel, 2009], in which the author clearly exposes the main contribution supplied by the Jersey library, i.e. the JAX-RS annotations, and guides the reader to build a simple Web Service supporting the CRUD operations (Create, Read, Update, and Delete) from scratch. Another influential reading, especially for guaranteeing a level of consistence among the HTTP codes returned upon the invocation of a specific resource, is "Using HTTP Methods for RESTful Services" by Todd Fredrich [Fredrich, 2012], part of "Learn REST: A RESTful Tutorial" by Pearson eCollege. The last, but not least important, building block for the server is *Apache Tomcat*[2], by the Apache Software Foundation: this is an HTTP web server and servlet container, capable of running web applications (like JaCO), accepting incoming HTTP connections from remote users and forwarding them to the servlet mapped to the requested URI for further handling.

6.1.1 JTLV implementation

JTLV [3], developed by Yaniv Sa'ar, is an implementation of the Temporal Logic Verifier [Pnueli and Shahar, 1996] written in Java, with some components written in C and accessed via the Java Native Interface (JNI) for dealing with the exponential complexity derived from the usage of Binary Decision Diagrams (BDDs).

JTLV has been one of the building blocks for the Web Service because, as shown in [De Giacomo and Sardiña, 2007], it is possible to find a solution for the behavior composition problem (i.e., synthesize a controller) in exponential time by resorting to a reduction to satisfiability in Propositional Dynamic Logic (PDL). Alberto Iachini, from Sapienza University of Rome, built on top on JTLV the *Composition* class, that is able to accept an SMV (Symbolic Model Verifier) file containing the description of the community (i.e., the collection of available behaviors) and the target, and give as output a structure that allows the user to construct all the possible valid compositions, if they exist. The method `doComposition()` present in this class is invoked upon the request of a new composition submitted by the user, then the output produced by the

[2]http://tomcat.apache.org/
[3]http://jtlv.ysaar.net/

computation is parsed and stored in an XML file, which will later constitute the body of the response sent by the server to the user who requested the composition.

6.1.2 Project organization

The source code of the JaCO Web Service is divided into several packages, whose dissertation will be the main topic of the this section.

The jtlv.client package

The Client package contains a simple class, the `Tester`, which is basically a client for the JaCO Web Service itself, written using the Client API [4] of the Jersey library used to implement the server. This client has been continuously modified following the development of the Web Service, and thus it has proven to be very useful to test the proper functioning of the classes and methods finalized just before. Obviously, there is no way to invoke the creation and the execution of the `Tester` class by means of HTTP requests.

The jtlv.composition package

The Composition package contains a single class, but probably one of the most valuable ones: the **Composition** class. As discussed in section 6.1.1, this class provided by (Nome) has the very important task to give as output the structure describing all the possible compositions obtained by combining the available behaviors in such a way that they collectively mimic the intended coordination expressed by means of the target behavior. The class has undergone some slight modifications to make it able to keep the name of the file in which the SMV description of the community is stored as a class field (set during the instantiation of the class), and to print the output produced in a file to be later parsed and processed instead of the standard console output.

[4]`http://jersey.java.net/nonav/documentation/latest/client-api.html`

The jtlv.model.jaxb package

The Model package contains the classes used to represent the various objects received from the user along with the requests or sent back as responses. The JAXB subname comes from the Java Architecture for XML Binding library, which is responsible to *serialize* and *deserialize* Java objects from/to XML and JSON files, that is used in the management of such objects within the web service.

The objects definitions contained in this package are:

- The **Behavior** class, which is used to model the transition system that describes the capabilities of each of the available components of the system, and also to represent the collective behavior the user strives to achieve (the target). Technically speaking, the Behavior class maintains as class fields a string containing the name of the component, and a list of **NodeWithTransitions** objects that form the transition system;

- The **ClientID** class, that simply represents the string which identifies and individual user of the web service;

- The **ComputedComputation** class, which encapsulates the composition calculated by invoking of the `doComposition()` method of the **Composition** class, and is contained in the **CompositionResponse** object to be returned to the remote user;

- The **CompositionResponse** class, which constitutes the object returned upon the invocation of the GET verb on the `/composition` resource, and comprises the current position of the request in the message queue and, if the composition has been computed, the deterministic transition system that models it;

The jtlv.server package

The Server package contains all the Java classes that are in charge to receive requests coming from remote users, retrieve or compute the needed resource(s), and send them back attached to the response messages flowing through the Internet. Each of the API endpoints (*auth*, *behaviors*, *behavior*, *target*, and *composition*) has a corresponding

`<EndpointName>Resource` which, by means of class methods marked with the annotations defined in the JAX-RS specification, manages to handle the various operations that the user can ask to be executed by sending an HTTP message with the appropriate verb.

The typical structure of an `<EndpointName>Resource` class is shown in the following source code example:

```java
package jtlv.server;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Request;
import javax.ws.rs.core.UriInfo;

@Path("/<endpointName>")
public class <EndpointName>Resource {

  @GET
  @Produces(MediaType.TEXT_XML)
  public XMLResource getEndpointResource()
  {
    // Code that retrieves the resource requested and returns it
    // This method returns an XML representation of the resource
  }

  @GET
  @Produces(MediaType.APPLICATION_JSON)
  public JSONResource getEndpointResourceJSON()
  {
    // Code that retrieves the resource requested and returns it
    // This method returns a JSON representation of the resource
  }

  @POST
  @Consumes(MediaType.TEXT_XML)
  public Response createNewResource(XMLResource res)
  {
    // Code that builds a new resource based on the data received
    // Response contains the status code of the operation
  }

  @DELETE
  public Response deleteResource()
  {
    // Code that deletes the selected resource
  }

  // Other methods
}
```

Each of these classes is built dynamically by the application server whenever an HTTP message reaches the correspondent endpoint, and is automatically destroyed as soon as the method appointed to handle the incoming request finishes its execution and the returns the requested resource or a failure error code.

The `/composition` endpoint is the only one that behaves in a slightly different manner: when a POST message is sent to this URI, the server does not immediately ask the Composition class to compute it and to return the output to the user; rather, the request is inserted at the end of a *request queue* that is implemented as a `BlockingQueue` object in the singleton class `Queue`. When the application server is started, besides the thread that continuously awaits for incoming HTTP connections and then dispatches them to the servlets responsible for their handling, another "worker" thread, whose execution flow is described in the `DaemonThread` class, is spawned. This thread endlessly checks whether the *request queue* is empty or not: in the latter case, it takes the request at the peak of the queue, creates and writes the SMV file with the description of the current composition problem instance, asks the Composition class to compute it, and parses the output returned into an XML file, which will later be acquired by the user with the last GET request. In the first case, the thread is simply put into sleep mode for

500 milliseconds; after this time has passed, the thread will start over its task.

6.2 Application Programming Interface (API) of JaCO

6.2.1 Request a client ID

URI	`http://jaco.dis.uniroma1.it/1/auth`
Method	GET
Returns	A `ClientID` object (see Section 6.3.1), whose client_id value should be supplied with future requests.

6.2.2 Managing behaviors

Send a new behavior

URI	`http://jaco.dis.uniroma1.it/1/{client_id}/behaviors`
Method	POST
Request body	A `Behavior` object (see Section 6.3.2).
Params	**client_id**: The client_id obtained via `/auth`.
Returns	**If a behavior with the same name already exists:** A *200 OK* HTTP message. Note that the behavior available on the server will not be modified; if you want to update the behavior, you should send the request using the PUT method. **If a behavior with the same name does not exist:** A *201 Created* HTTP message, containing the URI that refers to the newly created behavior on the server.

Get a behavior

URI	`http://jaco.dis.uniroma1.it/1/{client_id}/behaviors/{id}`
Method	GET
Params	**client_id**: The client_id obtained via `/auth`.
Returns	A `Behavior` object (see Section 6.3.2).
Errors	**404**: There is no behavior with name `{id}` on the server.

Update a behavior

URI	`http://jaco.dis.uniroma1.it/1/{client_id}/behaviors/{id}`
Method	PUT
Request body	A `Behavior` object (see Section 6.3.2).
Params	**client_id**: The client_id obtained via `/auth`.
Returns	A *200 OK* HTTP message.
Errors	**404**: There is no behavior with name `{id}` on the server.

Delete a behavior

URI	`http://jaco.dis.uniroma1.it/1/{client_id}/behaviors/{id}`
Method	DELETE
Params	**client_id**: The client_id obtained via `/auth`.
Returns	A *200 OK* HTTP message.
Errors	**404**: There is no behavior with name `{id}` on the server.

Get the behaviors' community

URI	`http://jaco.dis.uniroma1.it/1/{client_id}/behaviors`
Method	GET
Params	**client_id**: The client_id obtained via `/auth`.
Returns	A `Behaviors` object (see Section 6.3.3).

6.2.3 Managing the target

Send a new target behavior

URI	`http://jaco.dis.uniroma1.it/1/{client_id}/target`
Method	POST
Request body	A `Behavior` object (see Section 6.3.2).
Params	**client_id**: The client_id obtained via `/auth`.
Returns	**If a target behavior already exists:** A *200 OK* HTTP message. Note that the target behavior available on the server will not be modified; if you want to update the target behavior, you should send the request using the PUT method. **If a target behavior does not exist:** A *201 Created* HTTP message, containing the URI that refers to the newly created target behavior on the server.

Update the target behavior

URI	`http://jaco.dis.uniroma1.it/1/{client_id}/target`
Method	PUT
Request body	A `Behavior` object (see Section 6.3.2).
Params	**client_id**: The client_id obtained via `/auth`.
Returns	A *200 OK* HTTP message.
Errors	**404**: There is no target behavior available on the server.

Get the target behavior

URI	`http://jaco.dis.uniroma1.it/1/{client_id}/target`
Method	GET
Params	**client_id**: The client_id obtained via `/auth`.
Returns	A `Behavior` object (see Section 6.3.2).
Errors	**404**: There is no target behavior available on the server.

Delete the target behavior

URI	`http://jaco.dis.uniroma1.it/1/{client_id}/target`
Method	DELETE
Params	**client_id**: The client_id obtained via `/auth`.
Returns	A *200 OK* HTTP message.
Errors	**404**: There is no target behavior available on the server.

6.2.4 Managing the composition

Ask for the computation of the composition

URI	`http://jaco.dis.uniroma1.it/1/{client_id}/composition`
Method	POST
Params	**client_id**: The client_id obtained via `/auth`.
Returns	A `CompositionResponse` object (see Section 6.3.4), with the queue_number value set to the current queue position of the submitted request, and the composition object empty.

Get the computed composition

URI	`http://jaco.dis.uniroma1.it/1/{client_id}/composition`
Method	GET
Params	**client_id**: The client_id obtained via `/auth`.
Returns	**If a the composition is not yet ready:** A `CompositionResponse` object (see Section 6.3.4), with the queue_number value set to the current queue position of the submitted request, and the composition object empty. **If the composition is ready:** A `CompositionResponse` object (see Section 6.3.4), with the queue_number value set to 0, and the composition object containing the computed composition.

6.3 Data models

6.3.1 ClientID

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<client_id>
  9ck35lkn82as8navhvf99iuhqn
</client_id>
```

6.3.2 Behavior

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<behavior>
  <name>behaviorName</name>
  <finiteStateMachine>
    <state node="firstNode">
      <transition action="firstAction">
        <target>secondNode</target>
      </transition>
      <transition action="secondAction">
```

```
        <target>thirdNode</target>
      </transition>
    </state>
    <state node="secondNode">
      <transition action="secondAction">
        <target>thirdNode</target>
      </transition>
    </state>
    <state node="thirdNode">
      <transition action="thirdAction">
        <target>thirdNode</target>
      </transition>
      <transition action="fourthAction">
        <target>firstNode</target>
      </transition>
    </state>
  </finiteStateMachine>
</behavior>
```

6.3.3 Behaviors

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<behaviors>
  <behavior>
    <name>firstBehaviorName</name>
    <finiteStateMachine>
      <state node="firstNode">
        <transition action="firstAction">
          <target>secondNode</target>
        </transition>
        <transition action="secondAction">
          <target>thirdNode</target>
        </transition>
      </state>
      <state node="secondNode">
        <transition action="secondAction">
          <target>thirdNode</target>
        </transition>
      </state>
      <state node="thirdNode">
        <transition action="thirdAction">
          <target>thirdNode</target>
        </transition>
        <transition action="fourthAction">
          <target>firstNode</target>
        </transition>
      </state>
    </finiteStateMachine>
  </behavior>
  <behavior>
    <name>secondBehaviorName</name>
    <finiteStateMachine>
      <state node="firstNode">
        <transition action="firstAction">
          <target>secondNode</target>
        </transition>
        <transition action="secondAction">
          <target>thirdNode</target>
        </transition>
      </state>
      <state node="secondNode">
        <transition action="secondAction">
          <target>thirdNode</target>
        </transition>
      </state>
      <state node="thirdNode">
        <transition action="thirdAction">
          <target>thirdNode</target>
        </transition>
        <transition action="fourthAction">
          <target>firstNode</target>
        </transition>
      </state>
    </finiteStateMachine>
  </behavior>
```

```
</behaviors>
```

6.3.4 CompositionResponse

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<compositionResponse>
  <queue_number>2</queue_number>
  <computedComposition>
      <isRealizable>true</isRealizable>
      <targetState stateName="a">
          <communityState>
              <behaviorState service="MyEnemyMech" state="NodeA"/>
              <behaviorState service="MyEnemyMineBot" state="NodeH"/>
              <behaviorState service="MyEnemyMineBot1" state="NodeT"/>
          </communityState>
          <transitions>
              <transition invoke="MyEnemyMech" action="MoveToNodeN"/>
          </transitions>
      </targetState>
      <targetState stateName="b">
          <communityState>
              <behaviorState service="MyEnemyMech" state="NodeN"/>
              <behaviorState service="MyEnemyMineBot" state="NodeH"/>
              <behaviorState service="MyEnemyMineBot1" state="NodeT"/>
          </communityState>
          <transitions>
              <transition invoke="MyEnemyMineBot1" action="MoveToNodeP"/>
          </transitions>
      </targetState>
      <!-- Code omitted for brevity -->
  </computedComposition>
</compositionResponse>
```

6.4 Usage scenario for JaCO

During the design and the development of the JaCO Web Service, we tried to imagine what the most likely interaction would look like from the perspective of the users requesting services from our program, and strived for making this communication as simple as possible.

The scenario we have in mind is the one in which, initially, the user requests the `/auth` service, and obtains as a response an alphanumeric string called `client_id`, which identifies the portion of the storage space in which the server collects all the information that the user will supply within the following requests, and thus it is mandatory that the `client_id` value must be sent as a parameter along to all the next invocations.

Subsequently, the remote user communicates all the behaviors involved in the desired composition by way of a collection of POST requests to the `/behaviors` endpoint, indicating in the body of each HTTP message the `Behavior` object containing the name and the specification of the transition system associated with the behavior subject of the request. Since no other message exchange occurred between the `/auth` request and these ones, all the invocations should return a *200 OK* message, stating that the server received the object and stored it within its storage without errors.

Afterwards, the next information the user should supply is the target behavior he desires the composition to realize; this is done by attaching the target, enclosed in a `Behavior` object as done before, to a POST request pointed to the `/target` Uniform Resource Identifier. Even in this case, the user should receive as response a *200 OK* HTTP message, proving that the server fully understood the request and did all the work needed to handle it.

Finally, the user asks the server to compute the composition by sending a POST message with an empty body to the `/composition` endpoint; this will cause the server to insert the composition demand at the end of the message queue. From now on, the remote user can query the status of fulfillment of the request by issuing the GET method on the `/composition` URI: if it has not been executed yet, the server will reply with a `CompositionResponse` whose `composition` section will be empty and the `queue_number` will be set to the current position in the message queue; if, on the contrary, the composition has been calculated, the user will receive it encapsulated in a `CompositionResponse` object with the `queue_number` value set to zero.

Figure 6.1 illustrates, with the support of an UML Sequence Diagram, the succession of requests and responses that constitute the most common scenario we described in this section.

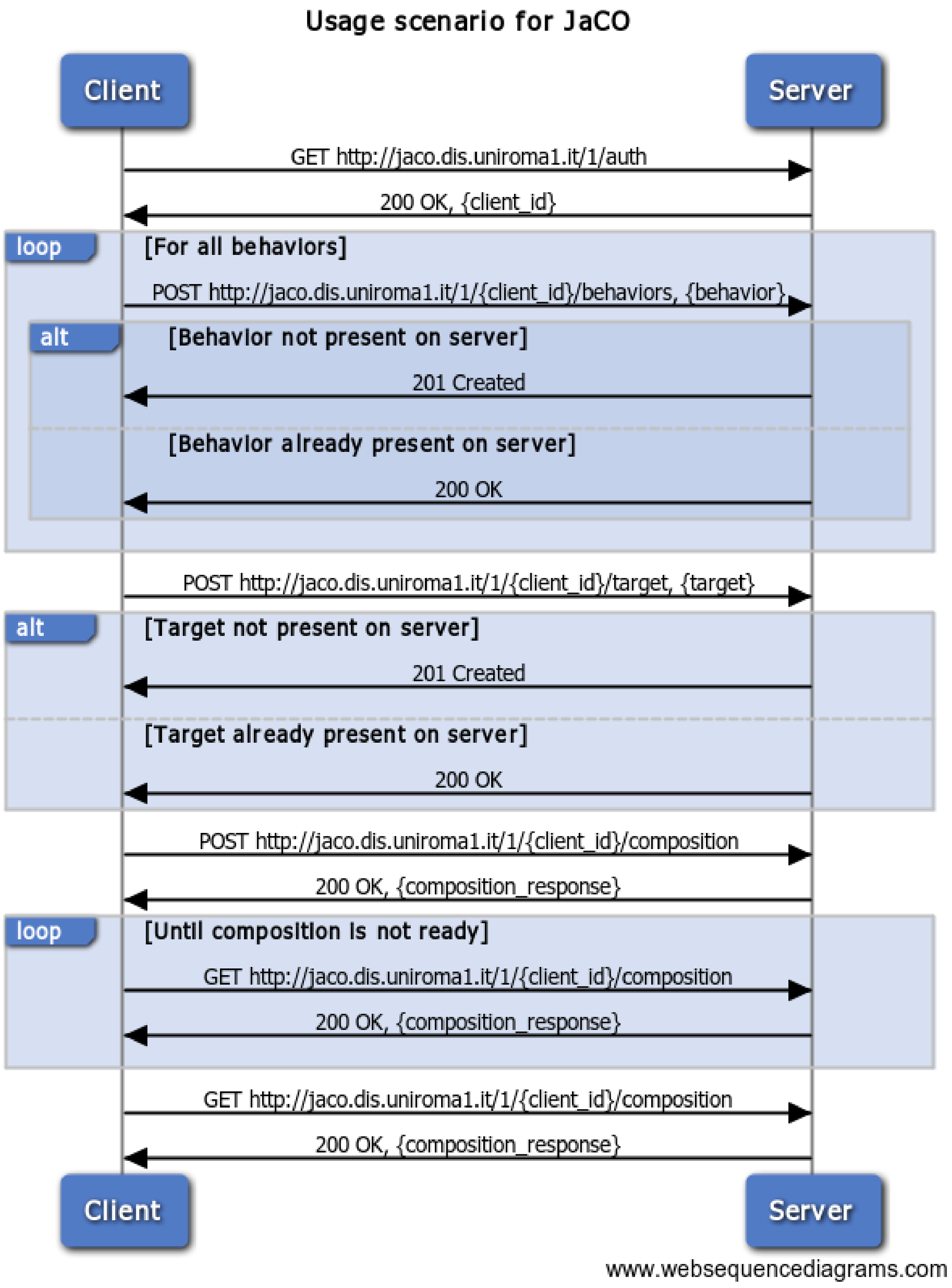

Usage scenario for JaCO
Client
Server
GET http://jaco.dis.uniroma1.it/1/auth
200 OK, {client_id}
loop
[For all behaviors]
POST http://jaco.dis.uniroma1.it/1/{client_id}/behaviors, {behavior}
alt
[Behavior not present on server]
201 Created
[Behavior already present on server]
200 OK
POST http://jaco.dis.uniroma1.it/1/{client_id}/target, {target}
alt
[Target not present on server]
201 Created
[Target already present on server]
200 OK
POST http://jaco.dis.uniroma1.it/1/{client_id}/composition
200 OK, {composition_response}
loop
[Until composition is not ready]
GET http://jaco.dis.uniroma1.it/1/{client_id}/composition
200 OK, {composition_response}
GET http://jaco.dis.uniroma1.it/1/{client_id}/composition
200 OK, {composition_response}
Client
Server
www.websequencediagrams.com

Figure 6.1: Typical usage scenario for the JaCO Web Service

CHAPTER 7

AN EXAMPLE OF USING JACO IN UNITY

7.1 JaCO in the Angry Bots patrolling domain

The test-bed for checking whether the novel web service was running and functioning properly has been to repeat the very same example described in Chapter 4. The example can be summarized as follows: in the environment in which the Angry Bots technical demo takes place, we identified a series of 20 points of interest (named with the letters of the alphabet from A to T); we placed in this environment three non-player characters, reusing the 3D models and the textures that came with the demo, and we gave to each NPC the ability of moving between these points of interest in such a way that no robot could cover all the points of interest by itself. We also added some overlaps on the NPCs' routes, i.e., there are some points of interest that could be reached by more than one robot. The coordination we want to achieve, i.e., the target behavior, is to put the user in charge of selecting the location he is interested to control and our behavior composition system to handle which NPC will realize the request. In particular, the controller provided by our system can select the right behavior to perform the chosen action, guaranteeing that the execution will not violate the properties that make the composition valid, for all future states.

The first experiment was conducted using the **cURL** [1] software, which is a simple but powerful command-line utility for transferring data with URL syntax, supporting a large number of different protocols and running on various platforms. The steps involved in realizing the usage scenario for the JaCO Web Service discussed in Section 6.4 with cURL are the following:

- Retrieving the client_id necessary for the subsequent interactions: the complete syntax of the command to be written in the console (or the command prompt) is
  ```
  curl http://jaco.dis.uniroma1.it/1/auth --request GET
  ```
  this request will return a `client_id` object, as described in section 6.3;

- For each of the available non-player characters, send the correspondent `Behavior` object to the server: this can be achieved by issuing on the command prompt
  ```
  curl http://jaco.dis.uniroma1.it/1/{client_id}/behaviors
  --request POST --header 'Content-Type:text/xml' --data
  @<behavior>.xml
  ```
  as many times as the number of available behaviors, assuming that the `Behavior` objects are stored in XML files present in the current working directory;

- Send the target behavior to the server: the command involved for this step is
  ```
  curl http://jaco.dis.uniroma1.it/1/{client_id}/target
  --request POST --header 'Content-Type:text/xml' --data
  @<target>.xml
  ```
  where `<target>` is the name of the XML file that contains the target behavior encoded as a `Behavior` object (as shown in Section 6.3);

- Ask the server to compute the composition based on the target behavior and the available behaviors sent in the previous requests: this can be achieved by simply sending a POST HTTP message with empty body to the `/composition` endpoint

[1] `http://curl.haxx.se/`

of the API, and the syntax of the command to be entered into the console is

```
curl http://jaco.dis.uniroma1.it/1/{client_id}/composition
--request POST
```

- Query the server about the current processing state of the submitted composition request: to get the `CompositionResponse` object, that gives the user information about the current position of the request in the queue maintained at the server or, if it has been computed, the requested composition, the user should issue the command

```
curl http://jaco.dis.uniroma1.it/1/{client_id}/composition
--request GET
```

After having conducted successfully the example by using the cURL software, we decided to extend the framework we developed for the Unity game engine to be able to send and receive the HTTP requests needed to realize the behavior composition directly within Unity, in a completely transparent manner to the user. For achieving this purpose, we relied on an open source C# library called **RestSharp** [2], which provides a simple but effective Application Programming Interface that allows the developer to build a web request, set all the necessary HTTP headers and parameters, issue the request to the server and retrieve the correspondent response, in both a synchronous and asynchronous way.

For example, the C# code needed to request the invocation of the flickr.photos.search method of the Flickr API [3], that returns a list of photos based on some criteria, with the RestSharp library would be the following;

```
var client = new RestClient("http://api.flickr.com");

var request = new RestRequest("/services/rest/", Method.GET);
request.AddParameter("method", "flickr.photos.search");
request.AddParameter("api_key", "3d93ac85d70d8c6fbaebb8ceb6918fdc");
request.AddParameter("lat", "41.54");
request.AddParameter("lon", "12.27");
```

[2] http://restsharp.org/
[3] http://www.flickr.com/services/api/

```
request.AddParameter("radius", "2");
request.AddParameter("format", "json");

// execute the request
RestResponse response = client.Execute(request);
var content = response.Content;
```

By taking advantage of this software component, we developed a new class called `RESTfulClient`, whose methods encapsulate all the requests that the user could need to issue to the server for the accomplishment of the usage scenario described in Section 6.4. In particular, the methods defined within the `RESTfulClient` class are:

- `Authorize()`, that is responsible for contacting the `/auth` endpoint of the JaCO API and retrieve the client_id to be communicated along all the following requests;
- `SendBehavior(string behaviorXmlString)`, that issues the POST HTTP message needed for transmitting to the server the `Behavior` object that the user passes to the function as a string argument;
- `SendTargetBehavior(string targetXmlString)`, similar to the previous method but concerned with contacting the `/target` endpoint and transmitting the encoded target behavior of the system;
- `RequestComposition()`, which is in charge of sending the POST HTTP message that causes the server to accept the composition request in its task queue;
- `GetComposition()`, that the user can call in order to query the server about the processing state of the composition request, and eventually retrieve the desired composition.

The results achieved with this approach are essentially the same we had already obtained using the SM4LL Composition Engine: the ActionLookupTable script has been slightly modified to make it capable of parsing the XML Schema Definition used by the JaCO Web Service, which is a little different from the one employed by the SM4LL engine; after this adaptation, the script has been able to build the hash table containing the

<current state of the system, next actions available> entries required by the framework we developed for the visualization of the behavior composition concept within the Unity game engine, leaving untouched the great performances achieved in the retrieval time of useful entries of the hash table. What really differentiates the JaCO Web Service from the composition engine provided by the SM4LL Project is the transparency to the user and the ease of the interface available to the developers: when the framework for the visualization in Unity is launched, the RESTfulClient class takes charge of performing all the interactions with the server needed to get the composition calculated, probably without the user even noticing; at the same time, it would be easy for a developer to map the functions involved in the process to, for example, the buttons of a Graphical User Interface, that would make the exchange a little more interactive, but still easy for the user.

CHAPTER 8

CONCLUSIONS

In this master thesis, we have investigated the feasibility of the application of the behavior composition technique to a video game-like setting, to address some of the challenges that usually arise in this kind of scenario, such as coordinating the action of non-player characters for making them realize a global behavior. We also investigated the feasibility of proposing the behavior composition technique as a web service accessible through the REST architectural style, to analyze whether it can represent a useful cloud-based tool for game developers in the near future.

In Chapter 4, The Angry Bots patrolling domain, we described all the classes belonging to the framework we developed for the visualization of the behavior composition concept within the Unity game engine, that made us achieve the second of the three objectives we set in the introductive chapter. In Chapter 6, Behavior Composition as a RESTful Web Service, we discussed the design and the implementation of the JaCO (Java-based Connection-Oriented) Web Service and explained the Application Programming Interface needed to interact with the service, that made us reach the first of the goals we aimed for. In Chapter 7, An example of using JaCO in Unity, we reported an example of application realized through the usage of the JaCO Web Service, which shows the potential utility arising from the application of the behavior composition technique to address challenges present in video game-like scenarios and the exposure of such method by means of a cloud-based tool for game developers.

The ideas and objectives that can be further investigated in future work arising from this master thesis are:

- Improve the source code of the Unity visualization framework, to make it easier to adapt for developing and running applications employing behavior composition within an environment different from the Angry Bots patrolling domain described in this thesis;
- Develop a sketch of interactive storytelling framework, in which the collection of available behaviors represents non-player characters that populate the world in which the story unfolds and propose quests to the player, the target behavior represents the possible branches that the storyline can follow based on the "alignment" (good or evil) that the player can choose by accepting different kinds of quests proposed, and the resulting composition forms the basis of the decision making process for a "storyline AI director", that proposed to the player different quests based on the branch of storyline selected and, at each point, check whether the player can ultimately pick up the final quest that takes him to the ending of the game.

BIBLIOGRAPHY

Bartheye, Olivier and Éric Jacopin. "Connecting PDDL-based off the shelf planners to an arcade game". In: *AI in Games Workshop at ECAI-08*. Patras, Greece, July 2008.

Berardi, Daniela, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Massimo Mecella. "Automatic composition of e-Services that export their behavior". In: *Proceedings of the First Interational Conference on Service Oriented Computing*. Trento, Italy, 2003.

Berardi, Daniela, Diego Calvanese, Giuseppe De Giacomo, and Massimo Mecella. "Automatic composition of e-Services that export their behavior". In: *Proceedings of the Thrid Interational Conference on Service Oriented Computing*. Amsterdam, The Netherlands, 2005.

De Giacomo, Giuseppe, Fabio Patrizi, and Sebastian Sardiña. "Automatic Synthesis of a Global Behavior from Multiple Distributed Behaviors". In: *Proceedings of the Twenty-Second Conference on Artificial Intelligence*. Vancouver, Canada, Aug. 2007.

De Giacomo, Giuseppe and Sebastian Sardiña. "Automatic Synthesis of New Behaviors from a Library of Available Behaviors". In: *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*. Hyderabad, India, 2007.

Erol, Kutluhan, James Hendler, and Dana S. Nau. “HTN planning: Complexity and expressivity”. In: *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*. Vol. 2. 1994, pp. 1123–1128.

Fielding, Roy Thomas. “Architectural Styles and the Design of Network-based Software Architectures”. PhD thesis. University of California, Irvine, 2000.

Fikes, Richard E. and Nils J. Nilsson. “STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving”. In: *Artificial Intelligence* 2 (1971), pp. 189–208.

Fredrich, Todd. *Using HTTP Methods for RESTful Services*. 2012. URL: `http://www.restapitutorial.com/lessons/httpmethods.html`.

Hoang, Hai, Stephen Lee-Urban, and Héctor Muñoz-Avila. “Hierarchical plan representations for encoding strategic game ai”. In: *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05*. 2005.

Kalin, Martin. *Java Web Services: Up and Running*. O’Reilly Media, 2009.

Kelly, John-Paul, Adi Botea, and Sven Koenig. “Offline Planning with Hierarchical Task Networks in Video Games”. In: *Proceedings of the Fourth International Conference on Artificial Intelligence and Interactive Digital Entertainment AIIDE-08*. 2008.

Magerko, Brian and John E. Laird. “Mediating the tension between plot and interaction”. In: *Proceedings of AAAI Workshop Series: Challenges in Game Artificial Intelligence*. San Jose, CA, 2004.

Mateas, Michael and Andrew Stern. “Integrating Plot, Character and Natural Language Processing in the Interactive Drama Façade”. In: *Proceedings of the First International Conference on Technologies for Interactive Digital Storytelling and Entertainment*. Darmstadt, Germany, 2003.

Michael, Buro. “Call for AI Research in RTS Games”. In: *In Proceedings of the AAAI Workshop on AI in Games*. 2004.

Millington, Ian and John Funge. *Artificial Intelligence For Games - Second Edition*. Morgan Kaufmann Publishers, 2009.

Muñoz-Avila, Héctor and Todd Fisher. “Strategic Planning for Unreal Tournament Bots”. In: *In AAAI Workshop on Challenges in Game AI*. AAAI Press, 2004.

Ontanon, Santi, Kinshuk Mishra, Neha Sugandh, and Ashwin Ram. "On-Line Case-Based Planning". In: *Computational Intelligence* 26 (2010). ISSN: 0824-7935.

Orkin, Jeff. "Three States and a Plan: The AI of F.E.A.R." In: *Proceedings of the Game Developer's Conference (GDC)*. 2006.

Pnueli, Amir and Elad Shahar. "A Platform for Combining Deductive with Algorithmic Verification". In: *Proceedings of the Eighth International Conference on Computer Aided Verification*. New Brunswick, NJ, USA, 1996.

Sardiña, Sebastian, Fabio Patrizi, and Giuseppe De Giacomo. "Behavior Composition in the Presence of Failure". In: *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning*. Sydney, Australia, Sept. 2008.

Vogel, Lars. *REST with Java (JAX-RS) using Jersey*. 2009. URL: `http://www.vogella.com/articles/REST/article.html`.

Yadav, Nitin Kumar. "Implementation and Analysis of Behaviour Composition Problem using Simulation". MA thesis. Melbourne, Victoria, Australia: Royal Melbourne Institute of Technology, 2009.

ACKNOWLEDGEMENTS