SAPIENZA
UNIVERSITÀ DI ROMA

FACULTY OF INFORMATION ENGINEERING, INFORMATICS AND STATISTICS

Bachelor of Science in Engineering in

COMPUTER SCIENCE AND CONTROL ENGINEERING

Curriculum

COMPUTER SCIENCE

Coordinating Videogame Dialogue Systems through Behavior Composition

---

Supervisor
Prof. Giuseppe De Giacomo
Co-Supervisor
Prof. Stavros Vassos

Candidate
Daniele Riccardelli

Academic Year 2013/2014

Contents

Chapter 1

Introduction

The goal of this project is to present a new approach towards the implementation of dialogues in videogames based on *Behavior Composition*. In the following section we present a motivation behind this work, along with a review of some existing approaches to the problem. From a research perspective, such work could be located in the field of *Interactive Storytelling and Narrative*, which is a form of digital entertainment mostly related to videogames. In Chapter 2 we give a formal definition of Interactive Storytelling, and we describe the framework of *Behavior Composition*. In Chapter 3, we introduce a videogame scenario that in Chapter 4 we use to describe how the Dialogue System developed works. In Chapter 5 we show how the story unfolds in our mini-game in order to help the reader to better understand how *Behavior Composition* is being used in our solution. In Chapter 6 we sum up our work and we present a set of possible directions for future work.

1.1 Motivation

Dialogue Systems are ubiquitous in videogames nowadays, thanks to the increasing research effort towards Interactive Storytelling and the demonstrated interest for story-driven experiences from the public. A recent clue of this trend in the videogame industry can be evinced from the success of Interactive Adventures like *The Walking Dead* and *The Wolf Among Us*,

both from *Telltale Games*, and the unexpected decision of Co-Founder Ken Levine to shut down *Irrational Games* - company behind AAA titles like the *Bioshock* trilogy - in order to focus on "narrative-driven games for the core gamer that are highly replayable"[1].

Figure 1.1: A screenshot from *The Walking Dead*[2].

As the amount of computing power available on PCs, consoles and handheld devices constanly increases, as well as the importance of *Cloud computing* applied to game-related tasks [3], i.e. adaptive gameplay and heavy A.I. calculations, the customers are expecting gaming experiences where interactions with *non-player characters* (NPCs) are more and more credible and complex.

A classic example of player-NPC interaction is a dialogue. Dialogues are usually triggered getting close to an NPC or clicking on it, and most of the times (*Mass Effect* or *The Elder Scrolls*) consist in the AI offering a set of lines to the player to pick in order to select the preferred answer. How this *Question & Answers* (Q&As) set is selected highly differs from solution

[1] `http://www.gamasutra.com/view/news/211067/Ken_Levine_shutting_down_Irrational_Games_to_start_anew_at_TakeTwo.php`

[2] source:`www.gamasutra.com`

[3] `http://news.xbox.com/2013/10/xbox-one-cloud`

to solution, but a common and easy to program approach is to *hard-code* these sets into the NPC logic. Each non-player character may have its own Q&A set internally stored as a tree, or load a prebuilt one at runtime, and the chosen branch is either given by the player's choice or by the value of dialogue-related or global variables that can indicate the current state of the story or the mood of the NPC.

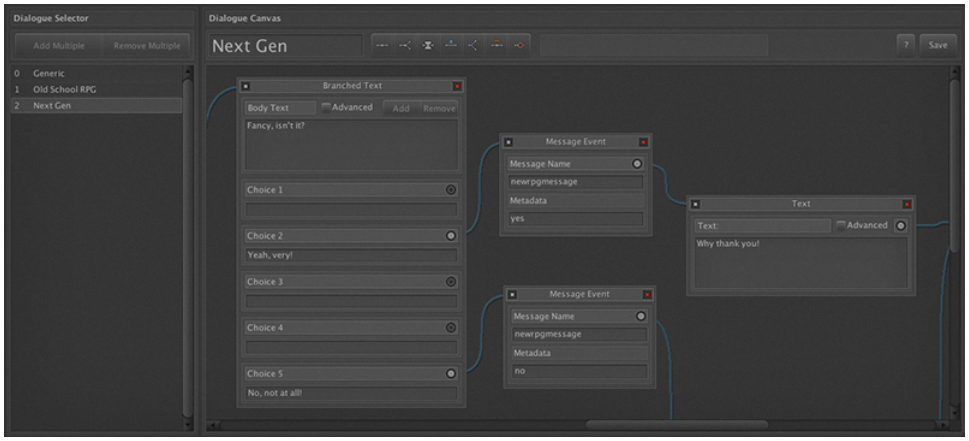


Figure 1.2: *Dialoguer*[4]: a Unity-based dialogue creator where dialogues are stored as branching conversation trees.

While this approach would be fast and effective for some games, it offers generally, from a developer perspective, very little flexibility in terms of story editing, and eventual story deadlocks would be hard to debug. In the following section we present a different, flexible approach towards the problem.

1.2 An AI flexible dialogue system

What if we consider the non-player characters as autonomous behaviors that concurr to fullfill an higher-level process, a *Target* behavior? This way, the game designer could focus on how the narration should evolve, in an intuitive *finite-state machine* (FSM) fashion, caring less about how this desiderd story flow is achieved, focusing instead on what he thinks would be fun for a player to experience.

[4] `http://www.dialoguer.info/`

Once he/she is done with this higher-level design, he could design and characterize the different NPCs that take part in the narration, and via a tool that we will describe later, compute (if it exists) a composition of such independent behaviors that fullfills the target flow of the story.
The Dialogue System that takes advantage of such composition is developed as a set of `C#` scripts that work as so-called *GameObject Components* for the *Unity* Game Engine[5].
The Components are designed in such a way that they can be attached as-they-are to the NPCs with no coding effort.

A more in-depth description of the solution is presented in next chapters.

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

Chapter 2

Literature review

2.1 Dialogue systems in videogames

What the dialogue system is, according to the player perspective, is an interface that facilitates an interaction between the player and an AI in a credbile way. This credibility is achieved from time to time in different ways, but mostly offering Q&As coherent to the current state of the story, the mood and characterization of the NPC and the player's choices up to that point. Moreover, a key factor in order to achieve such credibility and encourage immersion into the gaming experience, is to offer a range of interactions as broad as possible, so that different players with different approaches can be satisfied by a gaming experience that allows them to act the way they intend to. The effort of moving the narration according to the player's choices, as mentioned in Chapter 1, belongs to the scientific research field related to *Interactive Storytelling* and *Narrative*, and in the next section we are going to quickly review some of the relevant literature.

2.2 Interactive Storytelling

Interactive Storytelling is a branch of the AI whose research aims to develop interactive applications where the narration can be influenced in realtime by a user.[1] While it can be used for applications such as education, *Interactive*

Storytelling is generally linked to entertainment. The main challenge, from a researcher perspective, can be found in the effort to balance the desire for a coherent narration unfolding with the user agency. The user may, in fact, cross the boundaries the experience was designed to take place in, unveiling the system limitations, which in turn may start acting inconsistently.[2] Wrapping up the most relevant efforts related to *Interactive Narrative Systems*, in [2] they cathegorize such works considering three factors that build up a 3D space:

1. Authorial intent: how much does the author influence the narration? Consequently, how much does it limit the player's freedom of action?

2. Virtual character autonomy: How much are the virtual characters constraint by the narrative? The two extremes are *Strong Story* and *Strong Autonomy*, where the first implies that the virtual character acting is fully subordinated to the narration, while the latter implies that the virtual character is in full control of its actions.

3. Player modeling: How does the system adapt itself to the user?

To cite a famous research effort in the field of Interactive Storytelling, *Façade*[3] has been placed somewhere in the centre between *Strong Story* and *Strong Autonomy*, where a *drama manager*[4] provides coherence and NPCs are autonomous in the sense that they can individually decide how to accomodate the desired higher-level narration flow.

When talking about a *drama manager*, we generally refer to an omniscent agent which monitors and influences the characters in the scene in order to comply with a desidered set of constraints, which can be, for instance, related to the coherence of the story presented or linked to some parameters that in some way can cathegorize the quality of the interactive experience.
According to the same classification, an approach based on JaCO - the web service we'll talk more about in the following chapters - instead, in regards to the *Virtual character autonomy*, can be placed closer to the *Strong Story* end. This is because the characters populating the scene have to obey to

their *individual Behavior*, along with complying with the constraints set by the *Target Behavior* at runtime.

2.3 Behavior composition

Behavior Composition aims to realize a *Target process* through the coordination of a set of available *Behaviors*. The problem of *Behavior Composition* can be easily applied to entertainment applications such as videogames, where the Behaviors are formal descriptions of non-player characters and the Target process is the desired high-level unfolding of the story the game designer is interested to offer to the player.

Directing the reader to [5] for a more detailed description, we'll now present the *framework* of *Behavior Composition* - as it is crucial in order to understand the mechanisms that move JaCO, which is a fundamental component of the Dialogue System that will be introduced in the following chapters.

Such framework is based on the modeling of the *Behaviors* as transition systems, that serve as an input for the computation of a finite-state machine - the *Controller Generator* (CG) - that is able to generate at runtime any possible composition.

The *Behavior* is defined as a tuple $B = (S, s_0, A, \delta)$, where: 1. S is the finite set of behavior's *states*; 2. $s_0 \in S$ is the *initial state*; 3. A is the finite set of behavior's *actions*; 4. $\delta \subseteq S \times A \times S$ is the behavior's *transition relation*. Initially, the non-player character who is paired with a certain *Behavior* starts in the initial state s_0; from current state s, the character can execute any action a "labeling" some transition outgoing from s, i.e., s.t. $s \xrightarrow{a} s'$, for some $s' \in S$; after the action is executed, the NPC nondeterministically moves to a successor state s'' s.t. $s \xrightarrow{a} s''$, which becomes the next current state. From the new state, a new action may be requested, which leads to a new iteration. We assume given a finite set of NPCs, each described by a behavior $B_i = (S_i, s_{i0}, A_i, \delta_i)$, and denote the obtained set as $P = \{B_1, \ldots, B_n\}$. We de-

fine the set $A_P = \bigcup_{i=1,\dots,n} A_i$, containing all the actions occurring in some behavior of P. Given P, a so-called *target* behavior T is a behavior $T = (S_t, s_{t0}, A_t, \delta_t)$, s.t., $A_t = A_P$.

The goal of behavior composition in videogame applications is to exploit the NPC Behaviors in order to *realize* the decisions made by a drama manager at runtime. This boils down to select a Behavior that is able to realize the action *currently requested* and that makes possible for all the future requests compliant with the *Target Behavior* to be executed.
In [5], a solution technique for the problem is presented. Such solution returns a FSM, called the *Controller Generator* (CG), able to generate, at runtime, *any* existing composition.
This CG can be shaped as a table, and looked up at runtime at each decision point of the drama manager.

Chapter 3

"Uncommon crime scene" Scenario

3.1 Overview

The best way to demonstrate how this **Dialogue System** works, and how different it is, compared to the existing solutions, is by showing a simple demo that borrows some of the narrative patterns followed by most of the commercial adventure videogames.

This demo can be classified as a dialogue-based adventure 3D game. The choice to develop a demo belonging to this genre is due to the natural representation of the unfloding of the story through **f**inite **s**tate **m**achines (**FSM**).

The player is a detective who has to investigate a crime which just took place in a desolated building. The goal of the game is to find out who's the criminal among the characters populating the scene. This is achieved questioning them one by one.

The player, on one hand, can interact with the characters by approaching the specific NPC he is interested in talking to. The NPCs, on the other hand, are not interested in starting the conversation with the player themselves, and

walk nervously around the world map. The allowed interactions consist of a set of questions that pop up when the player is close enough to a non-player character and vary according to the interested NPC state and the current state of the story. By selecting one of them, therefore making a *choice*, the NPC will answer consistently.

The consequence of an interaction varies from choice to choice, where some have no effect at all and can be compared to *filler* game quests in commercial games, where the player might know more details about the story background or have some character's insight, that would make the playing more interesting and immmersive, but still doesn't add up progress to the *main* flow of the story, which is the path the game designer is mostly interested for the player to take. Other quests, instead, can alter the *internal state* of the NPC. What an internal state can mean highly depends on the game or on the NPC itself: it can be a *mood*, because, maybe, annoying him/her with some specific utterance, he/she will interact with us in a complete different (and coherent) way (as it happens in the demo here presented); otherwise, it can be a *faction*, in war games, and this is intuitive to figure out. Summarizing, the high-level description of the evolution of the NPC throughout the game is script related, but these *behaviors* all lead to a similar FSM-shaped low-level representation.

Intuitively, there are also interactions that influence the path taken by the story, and can be compared to *main quests*, and these can and cannot, varying from quest to quest, influence the state of the NPC who facilitates them. Finally, the game will feature *story branching*, which means that there will be different (two, in this case) paths to get to the end of the game. In commercial games, this is used to simulate a more granular control by the user over the unfolding of the story and to allow some level of *replayability*.

In the following sections will be presented an in-depth description of the game scenario.

3.2 Storyline interactions

The dialogues between Player and AI are modeled in a way that an atomic interaction consists of a question and answer pair.

It is worth mentioning that these lines, unlike in other games, are not NPC-related, or *hardcoded* into the NPC logic. Instead, they belong to a pool of interactions that is queried at runtime.

3.2.1 Q & A set

Q1: Hey there, what’s going on?

A1: Finally! There is a terrible thing that just happened here! Go inside and investigate! Quick!

A2: Are you still here?!? Run inside! The criminal might well still be around!

Q2: You, little kid: do you know anything about this crime?

A1: I could tell you...if only you could give me something in return...first.

Q3: The little kid is looking for something hes lost. Do you know what that is?

A1: Oh, I guess I do! I found this in the yard, this morning.

A2: Hmm, I have no idea. I barely see that kid around.

Q4: Here you are. Can you tell me now?

A1: If I were you, I would ask Mrs. White over there

Q5: Confess! Its you the one who committed the crime!

A1: I dont know what youre talking about! I dont even like cookies!

A2: Prove it, you disrespectful cop!

A3: Ahah, nice try, my friend!

Q6: Sorry! It might well not be you after all

A1: Obviously not! Youre accusing the wrong person! Mrs. Pink, SHE was acting suspiciously!

Q7: What can you tell me about the woman in the pink dress?

A1: I saw her running away as I heard that scream...luckily Mr. Brown stopped her.

A2: Now that youve asked...isn't she hiding something in her pocket?

Q8: This place...do things like these happen often here?

A1: You know, dark streets and no police around make it easy for criminals to operate.

A2: Not at all, I wouldnt say so.

Q9: Mrs. White looks overly nervous. What can you tell me about her?

A1: Let's say: I don't think she has anything to do with this crime.

A2: I saw her from behind the window, there in the kitchen, just before it turned into the crime scene.

Q10: It was you! I asked witnesses! CONFESS!!!

A1: Oh no! Yes, it was me...but I swear, that was just a cookie!

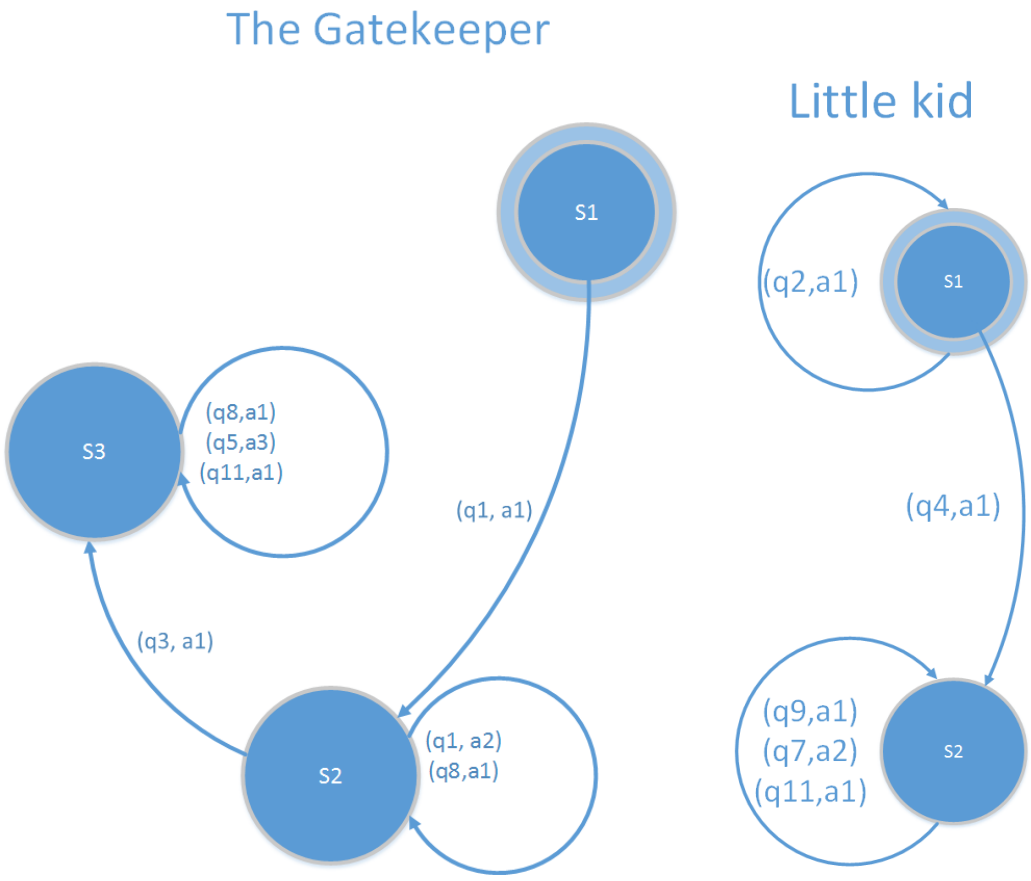
Q11: *proud air, whistling*

A1: Good job, detective!

A2: Im not gonna go to jail, am I?

3.3 Non-player characters

The characters that will populate the game scenario will be, player aside, five, differently characterized and moved by the following finite-state machines:

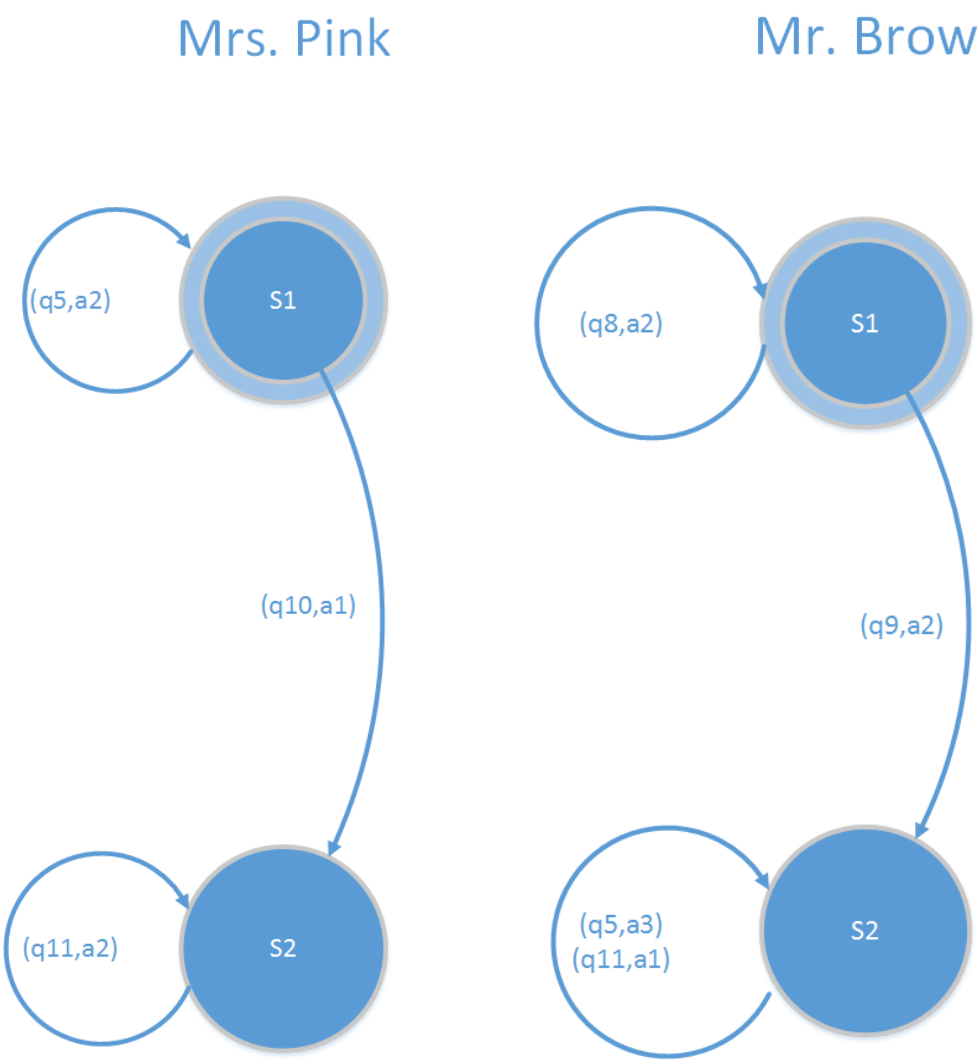


(a) The Gatekeeper (b) Little kid

Figure 3.1: The Gatekeeper 3.1(a) and Little Kid 3.1(b) FSMs

The Gatekeeper: the first character met thorughout the game, which acts as an introductive character that explains to the player the game setting and what's their task. In fact, no interaction is allowed with other characters unless the player first interacts with the Gatekeeper.

Little kid: A key character for the unfolding of the story, since he's one of the witnesses of the crime.



(a) Mrs. Pink (b) Mr. Brown

Figure 3.2: Mrs. Pink 3.2(a) and Mr. Brown 3.2(b) FSMs

Mrs. Pink: She's the one who turns out to be the criminal. The player can accuse her soon, but he/she will need a stronger proof to let her confess.

Mr. Brown: He's not crucial for completing the story, but offers more substance to the narration, and, most importantly, the player can't deduce his role being secondary until the very end.

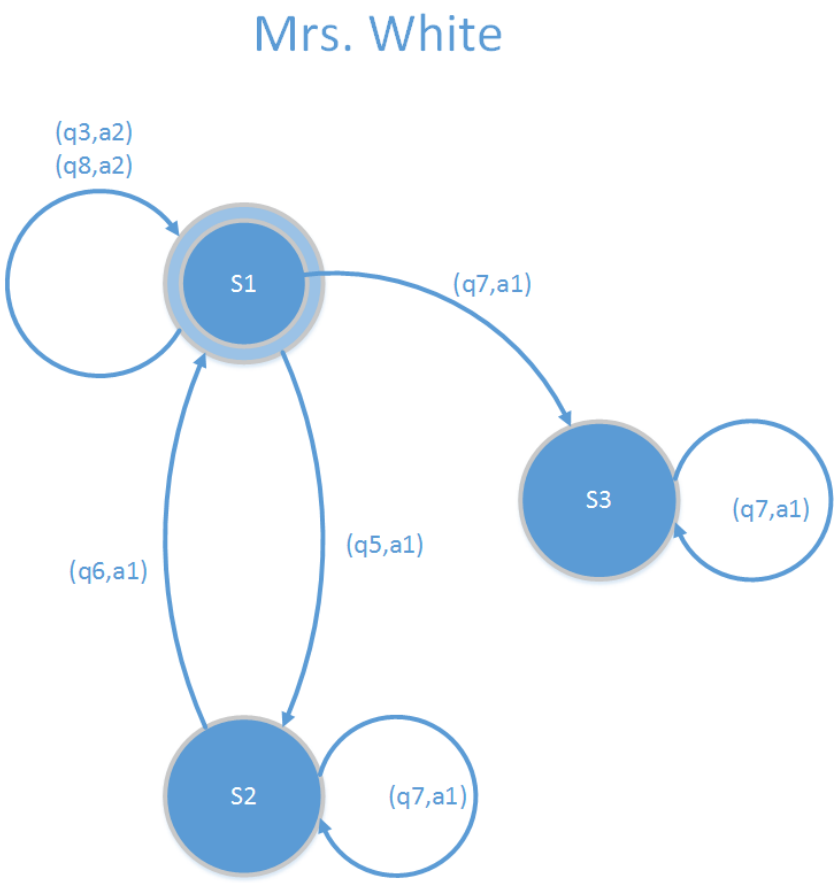


Figure 3.3: Mrs. White

Mrs. White: This character shows how the Behavior FSM can model the mood of an NPC. Her presence grants the game to be completed via two different story branches: either trusting or upsetting her.

3.4 Intended storyline

For any interaction and decision should the player decide to take, it is granted that it will conform to the *Intended storyline*. This is a finite-state machine written at design time that specifies what are the high-level paths allowed

for the player to take throughout the story. The importance of modeling the intended storyline as a FSM is twofold: i) it introduces a simple way to trace the player desired progress into the story; ii) the game designer can make sure that the player will experience all the game steps that he considers crucial and/or fun to play.

In Figure 3.4 it is clear how the atomic interactions consist of (question, answer) pairs, and how some quests are crucial for the unfolding of the story, while others keep the story in the same state. Also, one could attribute each state *S#* with some more meaningful tag: in this case, for example, being in *S1* could mean *the player just started the game*, while being in *S2* could mean *the player got introduced to the crime scene*, etc...

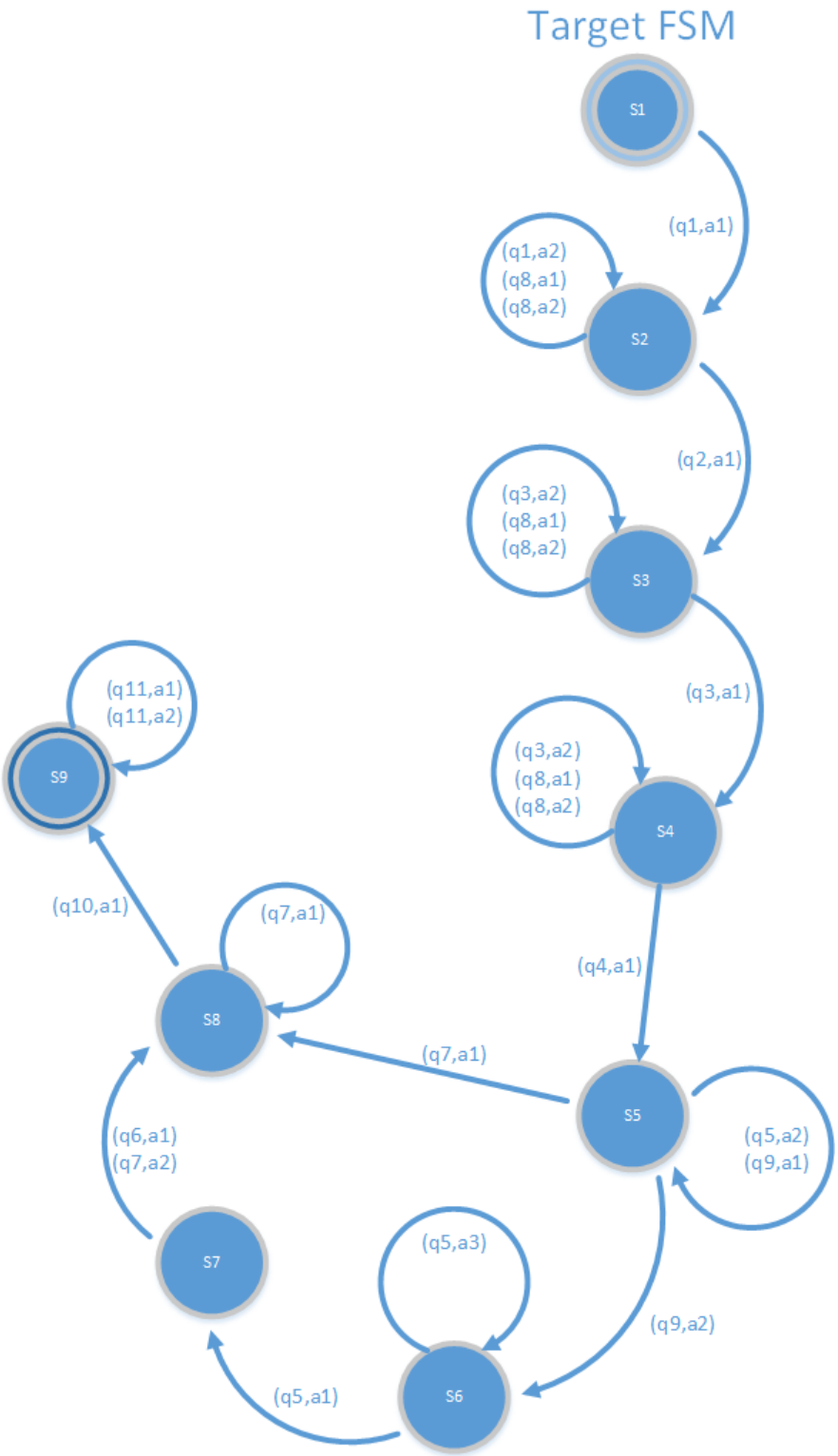


Figure 3.4: Representation of the desired story flow as a FSM

3.5 Behavior Composition problem

How is this scenario built though? How can one grant that, given a set of NPCs with individual behaviors, all the interactions defined in the Target FSM are executable for each possible game run? Failure to do so would lead, in some cases, to game dead ends, where the player is not capable anymore of completing the game. Furthermore, such problems can be hard to identify and debug, as the developer might have to test all the possible story branchings considering all the individual NPC and Target states.

This problem can be solved computing a *controller*, which has been described in the *Literature* section. The existance of such controller, for a given set of Behaviors and Target FSMs, is important for two reasons: looking it up at runtime, we can easily select the relevant interactions given a certain game context (NPC and Target state); moreover, its existance grants that, for each game run (i.e. each set of player choices), the interactions defined at each state of the *Target FSM* can be facilitated, so no dead end can arise.

In the next chapter will be presented a deep description of how this scenario has been implemented in Unity, and which components it relies on.

Chapter 4

The "Uncommon crime scene" mini-game

In order to develop a mini-game based on the scenario described in the previos chapter, the game engine chosen for this task is **Unity**[1].Under the hood, most of the game logic is programmed in C# making use of the *Unity* Scripting API and some additional functionalities from **.NET**.

Fundamental components of this demo will also be the **.xml** files storing the *game script*, i.e. the questions and related answers that will be available during gameplay, as well as the .xml's that describe the *NPC behaviors*, the *Target behaviors*, and the computed *Behavior Composition*.

Lastly, at startup, the game will communicate with **JaCO**[2], a RESTful web service for *Behavior Composition*.

4.1 Unity

Unity is a multi-platform Game Engine which is getting more and more attention by academics and game developers because of its intuitive *Component-*

[1] `http://unity3d.com/`
[2] `http://jaco.dis.uniroma1.it/`

based nature, the quick prototyping workflow and for the possibility to ship to different platforms (Windows, Linux, Mac, Consoles, etc...) with most of the code being shared among the different builds.

Being *Component-based* means that each object in the *Scene*, which in Unity terms is a *GameObject* - is composed of different blocks - the *components* - which can influence the rendering of the object (transform, materials), or the behavior (scripts), and can be added simply by dragging and dropping them on the *GameObject* we are interested into. This means, for example, that if we plan to have some simple crowd behavior in our game, we can just author our *Random Walker* script and attach it to all the characters we want to act like this.

This flexibilty came in handy in the development of this mini-game, and in particular it highly simplified the design of the dialogue system this demo wants to show the efficacy of. In fact, one can use such dialogue system almost as-it-is, in a black-box fashion, adding the interested *Prefabs*[3] to the Scene. In regards to the NPCs populating the scene, little coding is required in order to let them act according to the *Behavior Composition*, as most of the work is done by the designer when writing the game script and building the *Behavior* and *Target* finite-state machines. Of course, the dialogue system here presented also adopts precise naming conventions the game developer needs to be aware of.

[3] A GameObject reusable in different game contexts.

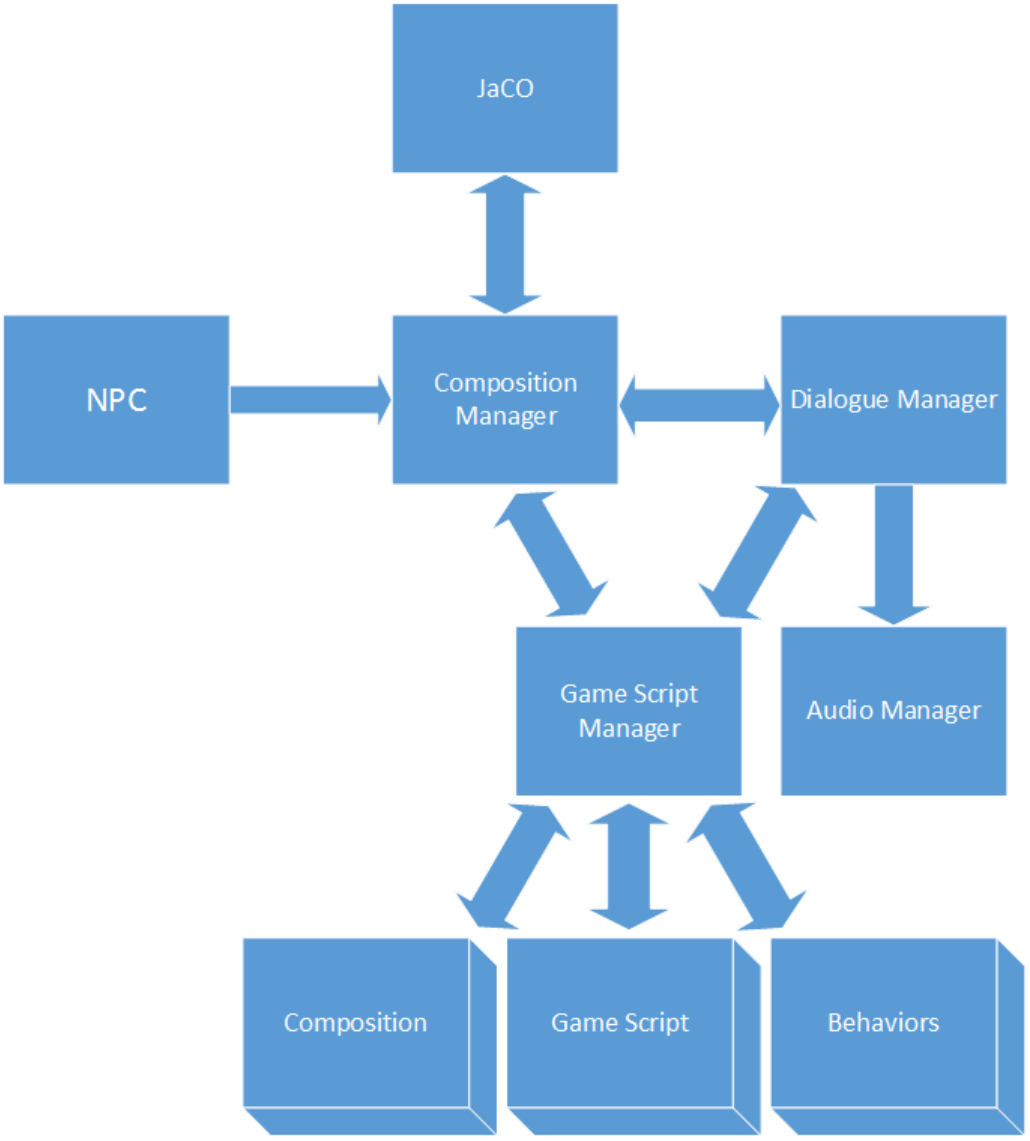


Figure 4.1: Interaction between all the components of the Dialogue System.

Before proceeding with an in-depth description of each system component developed, it is worth mentioning another reason why Unity is a top solution for research-related projects, which is the existance of the *Unity Asset Store*, where one can possibly grab all the art assets he needs to build up a simple game prototype and focus on the code and game mechanics.

This is exactly what happened with this demo: most of the art assets used

in this mini-game were taken from the free resources available on the *Asset Store*.

4.1.1 NPC

Programming the individual NPC logic is something that is mostly up to the game programmer, since different games have different requirements about how NPCs interact with the player and what they are able to do.

In this example, the NPCs are *random walkers* provided with a simple *WALK / TALK* finite-state machine, as showin in Figure 4.2:

Figure 4.2: General NPC FSM

They are assigned a set of *Wandar Points*, pick one of these randomly, reach the target through a path provided by the Unity built-in implementation of the $A*$ pathfinding algorithm, wait there a random amount of seconds (between the min and the max specified by the user), and then pick randomly another waypoint as the next destination.

The developer who is willing to use this dialogue system, though, has complete freedom to implement the game logic the way he prefers, as long as their NPC GameObject complies with the following constraints:

- The NPC GameObject name in the Scene must match the name of `.xml` file which describes its Behavior.

- The NPC logic script must inherit from the `Askable` abstract class, and call `requestInteraction(string NPCName)` and `dismissInteraction()`, implemented in the aforementioned base class, to start and end an interaction, respectively.
  This is the only way an NPC script should communicate with the Dialogue System, and gives the developer the possibility to use it as a black box.

- The NPC Behavior XML must reside in `Assets/Resources/GameXMLs/Behaviors/`, where `Assets` is the Asset folder of the project, and `Resources` is a subfolder that can be addressed when executing the game both in-editor and with standalone builds.

The following are code snippets from `Askable.cs` and `GeneralFSM.cs`, which is the file containing the NPC logic developed for this mini-game.

Listing 4.1: Code from `Askable.cs`

```
1  //from Askable.cs
2  using UnityEngine;
3
4  public abstract class Askable : MonoBehaviour{
5
6    public delegate void DialogueRequest(string NPCName);
7    public static event DialogueRequest OnDialogueRequest;
8
9    public delegate void DialogueDismiss(string NPCName);
10   public static event DialogueDismiss OnDialogueDismiss;
11
12   public virtual void requestInteraction(string NPCName){
13     OnDialogueRequest(NPCName);
14   }
15   public virtual void dismissInteraction(string NPCName){
16     OnDialogueDismiss(NPCName);
17   }
18 }
```

Listing 4.2: Code snippet from `GeneralFSM.cs`

```
1  //from GeneralFSM.cs
2  using UnityEngine;
3  using System.Collections;
4  using System.Collections.Generic;
5  using Pathfinding;
6
7
8  public class GeneralFSM : Askable {
9
10   //part of the code omitted for brevity
11
12   //called when an object collides with the NPC collider
13   void OnTriggerEnter(Collider other){
14     //if such object is the player, the NPC turn towards him and
15     //requests the interaction to the Dialogue System
16     if(other.gameObject.tag == "Player"){
17       playerPos = other.transform;
18       playerApproached = true;
19       Debug.Log ("requesting interaction");
20       requestInteraction(gameObject.name);
21     }
22   }
23
24   //called when an object leaves the NPC trigger collider
25   void OnTriggerExit(Collider other){
26     //if such object is the player, the NPC resumes its Random Walker
27     //behavior and asks the Dialogue System to dismiss the interaction
28     if(other.gameObject.tag == "Player"){
29       playerApproached = false;
30       Debug.Log ("dismissing interaction");
31       dismissInteraction(gameObject.name);
32     }
33   }
34 }
```

To give a broader exemple about what entities can benefit of such Dialogue System, one can consider modeling interactive objects, such as a chest, as

Behaviors. The chest starts from a *Locked* state. If the player approaches it, it will return a *Chest locked* answer. As the player progresses into the story, he can get to a point where he finally owns the key. At this point, the chest can offer the *Grab chest content* interaction. At this point, the chest could move to an *Opened* state, where any following interaction with the chest could lead to the *Empty chest* interaction.

4.1.2 Game Script

A crucial part during the game design process is to write the game script. With the approach here presented, the scripts lines are not hardcoded into the NPC logic, as it generally happens. Instead, they all belong to a pool of interactions, identified by pairs of questions and related answers.

The following is a snippet from the Game Script XML file:

Listing 4.3: Code snippet from `GameScript.xml`

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <dialogues>
3    <dialogue id = "1">
4      <player>Hey there, what's going on?</player>
5      <answers>
6        <answer id = "1">
7          <text>Finally! There is a terrible thing that just happened
               here! Go inside and investigate! Quick!</text>
8          <!-- this is a possible future work, but it's not yet
               implemented! -->
9          <consequences>
10           <consequence field = "unlockMainDoor" value = "true" ></
                 consequence>
11         </consequences>
12       </answer>
13       <answer id = "2">
14         <text>Are you still here?!? Run inside! The criminal might well
                still be around!</text>
15       </answer>
```

```
16      </answers>
17    </dialogue>
18    <dialogue id = "2">
19      <player>You, little kid: do you know anything about this crime?</
            player>
20      <answers>
21        <answer id = "1">
22          <text>I could tell you...if only you could give me something in
                return...first.</text>
23        </answer>
24      </answers>
25    </dialogue>
26    <!-- rest of the dialogues omitted for brevity -->
27  </dialogues>
```

Along with the simplicity to parse it, storing the dialogues in an XML file has the added benefit to be easily human readible and editable from a person with no programming background, which a game designer can be.

4.1.3 Composition Manager

The Composition manager is an invisible GameObject in the game scene with the `Composition Script` attached to it.
It is responsible of communicating with the *JaCO* web service and retrieving a *Computed Composition*, if it exists.
Furthermore, it represents the interface between the AI and the Dialogue Manager.
In the following code snippets it is shown how the `Composition Script` handles the requests from the AI:

Listing 4.4: Code snippet from `CompositionScript.cs`

```
1  //handling the OnDialogueRequest event
2  public void requestInteraction(string NPCName){
3    //if the player is talking to
4    //another NPC, return
5    if(dialogueGui.isGUIBusy())
6      return;
```

```
7     //set the name of the current NPC
8     //the player is interacting with
9     currentNPCInteracting = NPCName;
10    requestInteraction();
11  }
12
13  //handling the OnDialogueRestarted event
14  public void requestInteraction(bool interactionRestarted = false){
15    //find the possible transitions, according to the current NPC and
          Target states
16    XmlNode targetState = scriptXml.findCurrentTargetState(
          currentTargetState);
17    XmlNodeList possibleStates = targetState.SelectNodes("
          possibleStates/possibleState");
18    XmlNode possibleState = findCommunityState(possibleStates);
19    XmlNode transitions = possibleState.SelectSingleNode("transitions")
          ;
20    List<string> NPCTransitions = new List<string>();
21    findNPCTransitions(ref NPCTransitions, transitions);
22    //if the current NPC can interact with the player
23    //send these interactions to the Dialogue Manager
24    //otherwise disable Dialogue Input Mode
25    if(NPCTransitions.Count > 0)
26      dialogueGui.addInteractionOnGui(currentNPCInteracting,
            NPCTransitions, interactionRestarted);
27    else
28      OnNoInteractionAvailable();
29  }
30
31
32  //handling the OnDialogueDismiss event
33  public void dismissInteraction(string NPCName){
34    //if the event was fired by an NPC other
35    //than the one we are interacting with, return
36    if(currentNPCInteracting != NPCName)
37      return;
38    //hide the dialogue window, if it is visible
39    if(dialogueGui.isGUIBusy()){
40      dialogueGui.cleanGui(true);
```

```
41     }
42   }
```

Due to the way Unity manages *public* and *serialized* fields in a `C#` script, one can easily set up the script to work the way he wants to from inside the *Unity Inspector*:

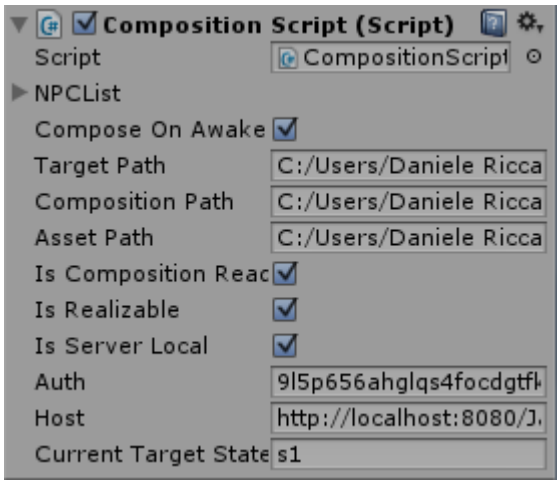


Figure 4.3: How the `Composition Script` looks on the Inspector after retrieving the computed Composition.

The following is a code snippet from `CompositionScript.cs` which shows what happens on game startup:

Listing 4.5: Code snippet from `CompositionScript.cs`

```
1 //from CompositionScript.cs
2   // Use this for initialization
3   void Start () {
4     //initalize the dictionary that stores the NPC state context
5     NPCStateContext = new Dictionary<string, string>();
6     //get a reference to the scripts CompositionScript communicates
          with
7     findScripts();
8     //if an NPC is missing in the game scene, print an error message
9     checkNPCsExistance();
10    //calculate the directory paths at startup
11    assetPath = Application.dataPath;
```

```
12      targetPath = assetPath + "/Resources/GameXMLs/Target/Target.xml";
13      compositionPath = assetPath + "/Resources/GameXMLs/Composition/
            Composition.xml";
14      //the server the script has to communicate with
15      host = isServerLocal ? "http://localhost:8080/JaCO/" : "http://jaco
            .dis.uniroma1.it/1/" ;
16      #if JaCO
17      //in case a new composition is required
18      if(composeOnAwake){
19        if(OnCompositionPhaseStarted != null){
20          //notify the subscribers that we are
21          //now communicating with the server
22          OnCompositionPhaseStarted();
23        }
24        askJaCO();
25      }
26      #endif
27    }
```

The complete list of *events* this script fires, along with the corrisponding *delegates*, is the following:

Listing 4.6: Code snippet from `CompositionScript.cs`

```
1  //communication with JaCO started
2  public delegate void CompositionPhaseStarted();
3  public static event CompositionPhaseStarted OnCompositionPhaseStarted;
4
5  //communication with JaCO ended
6  public delegate void CompositionPhaseEnded();
7  public static event CompositionPhaseEnded OnCompositionPhaseEnded;
8
9  //according to the Composition, the NPC can't interact with the player
10 public delegate void NoInteractionAvaiable();
11 public static event NoInteractionAvaiable OnNoInteractionAvailable;
```

The first two are related with the communication with the *JaCO* web service, while the third fires off to notify that the interested NPC cannot communicate with the player, therefore the input can return from *Dialogue Input Mode*,

where the player can use the mouse to interact with the *Dialogue GUI*.

4.1.4 Dialogue Manager

This invisible *GameObject* in the game scene is equipped with the `Dialogue Gui` script, which takes care of drawing the *Dialogue GUI* on screen. It receives from the the *Composition Manager* all the possible interactions that the NPC can offer to the player and shows them on screen, so that the player can interact with the *Dialogue GUI*, and select options to choose.

A nice thing of the *Unity* GUI System is that ease of switching GUI Skins, which define the way the GUI looks.
The Dialogue System here presented takes advantage of this feature, so that the designer can easily drag and drop the skin he likes the most into the `Dialogue Gui` inspector. Eventually, he can swap in and out new skins at runtime to better compare the way they look.

Figure 4.4: `Dialogue Gui` Inspector

In the mini-game developed here, we are replacing the Standard GUI with a custom so-called *Necromancer GUI*, which is a free asset available on the *Asset Store*.

The following is a code snippet from `DialogueGui.cs` :

Listing 4.7: Code snippet from `DialogueGui.cs`

```
1 //iterate through all the possible interactions
2 foreach(KeyValuePair < KeyValuePair <int, int> , KeyValuePair <string,
      string> > indexNText in resultingIndexedLines){
3   //if the player selects one of them...
4   if(GUILayout.Toggle(false, indexNText.Value.Key)){
5     //notify all the subscribers
6     OnChoiceSelected();
```

```
7     //prepare the script to the next frame
8     isQuestionSelected = true;
9     correctList(indexNText.Key);
10    //update NPC and Target states
11    KeyValuePair <string, KeyValuePair <int, int>> selectedTransition =
12      new KeyValuePair<string, KeyValuePair<int, int>>(NPCName,
            indexNText.Key);
13    string NPCDestination = scriptXml.getTransitionDestination(
          selectedTransition);
14    Debug.Log(NPCName + " next state: " + NPCDestination);
15    string targetDestination = scriptXml.getTargetDestination(
          indexNText.Key);
16    Debug.Log("Target next state: " + targetDestination);
17    KeyValuePair <string, string> NPCStateKvp = new KeyValuePair<string
          , string>(NPCName, NPCDestination);
18    updateStates(targetDestination, NPCStateKvp);
19  }
20  GUILayout.Space(4);
21 }
```

As the code may suggest, all the machinery related to dealing with the XML files is hidden to the `Dialogue Manager` and left to the `Game Script Manager`, which then returns the data necessary to udate NPC and Target states.

The complete list of *events* this script fires, along with the corrisponding *delegates*, is the following:

Listing 4.8: Code snippet from `DialogueGui.cs`

```
1 //the player selects a dialogue choice
2 public delegate void ChoiceSelected();
3 public static event ChoiceSelected OnChoiceSelected;
4
5 //the player is willing to continue their dialogue with the NPC
6 public delegate void DialogueRestarted(bool interactionRestarted =
      false);
7 public static event DialogueRestarted OnDialogueRestarted;
8
9 //the Dialogue Gui appears
```

```
10 public delegate void WindowAppeared(bool interactionRestarted = false);
11 public static event WindowAppeared OnWindowAppeared;
12
13 //the Dialogue Gui disappears
14 public delegate void WindowDisappeared();
15 public static event WindowDisappeared OnWindowDisappeared;
```

Apart from the interaction with the `Composition Manager`, an *event-based* approach makes it easy to plug an audio system in, which is what we'll discuss next.

4.1.5 Audio Manager

This *Audio Manager*, like the aforementioned *Game Script Manager*, *Composition Manager* and and *Dialogue Manager*, is an invisible *GameObject* in the scene, which is, in this case, equipped with an *Audio Source* component and a `C#` script named `Audio Script`.

The *Audio Source* component is a standard component Unity uses to play 3D sounds in the game space.

Since *Audio Manager* manages 2D sounds, such as sound effects related to GUI events, it was placed in the *Hierarchy* panel as a player character child object. This way, the sound heard by the player will not be distance and direction dependent as it will follow its parent object and keep its pitch wherever the player is.

The way `Audio Script` works is pretty simple: upon being notified of a GUI *event* it is interested into, it plays an audio clip accordingly.
Like the GUI Skin in the `Dialogue Gui` component, the audio clips can be added via the `Audio Script` *Inspector* panel and, eventually, changed at runtime.

4.2 JaCO

JaCO (**Ja**va-based **C**omposition-**O**riented Web Service) is a RESTful web service for orchestrating agents and devices whose behavior is expressed as a finite state machine (FSM).[4]

This means that, modeling the NPCs populating the scene as agents whose behavior is expressed as an FSM, one can make use of such web service to compute a controller that at any state of the story can tell which NPC can facilitate a certain interaction.

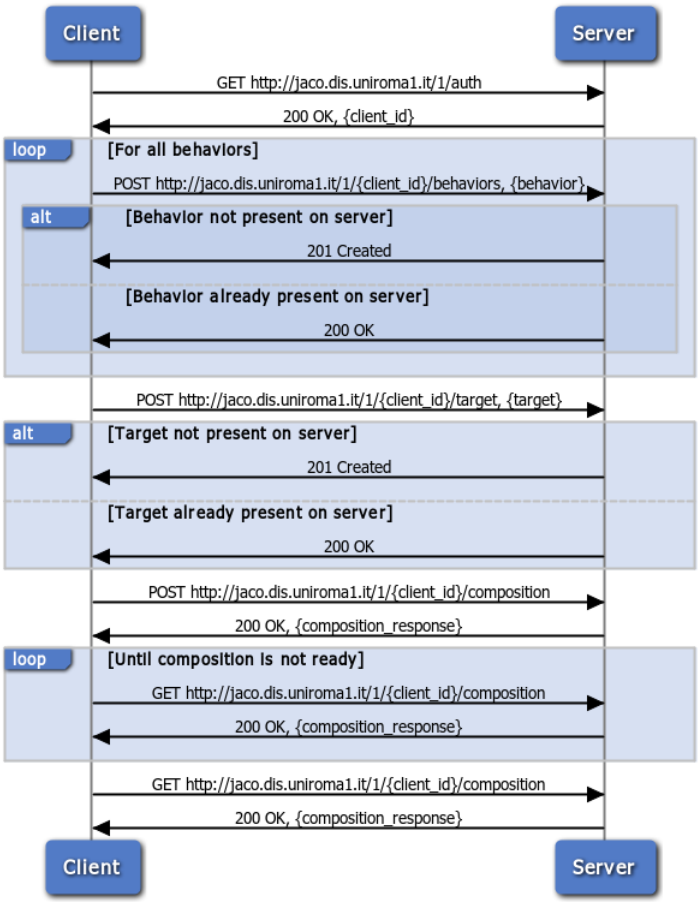


Figure 4.5: JaCO usage [5]

[4] `http://jaco.dis.uniroma1.it/`

Referring to the FSM shown in Figure 3.1(a) related to the *Gatekeeper* NPC, the following is the corresponding `TheGatekeeper.xml` file which expresses the same information in a way that conforms to the JaCO input requirements:

Listing 4.9: Code from `TheGatekeeper.xml`

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<behavior>
    <name>TheGatekeeper</name>
    <finiteStateMachine>
        <state node="s1">
            <transition action="q1a1">
                <target>s2</target>
            </transition>
        </state>
        <state node="s2">
            <transition action="q1a2">
                <target>s2</target>
            </transition>
            <transition action="q8a1">
                <target>s2</target>
            </transition>
            <transition action="q3a1">
                <target>s3</target>
            </transition>
        </state>
    <state node="s3">
      <transition action="q8a1">
                <target>s3</target>
            </transition>
      <transition action="q5a3">
                <target>s3</target>
            </transition>
            <transition action="q11a1">
                <target>s3</target>
            </transition>
    </state>
    </finiteStateMachine>
</behavior>
```

[5]Source: `http://jaco.dis.uniroma1.it/`

Of course, the other NPCs, while implementing their own individual behaviors, comply to the same structure. The `Target.xml` file is no exception, being itself the expression of a *Behavior*.

As mentioned in Subsection 4.1.3, if requested, `Composition Script` communicates with the JaCO web service (locally or remotely, according to developer's selection in the `Composition Script` Inspector panel) on startup. This happens as depicted in Figure 4.5, through **POST** and **GET** requests.

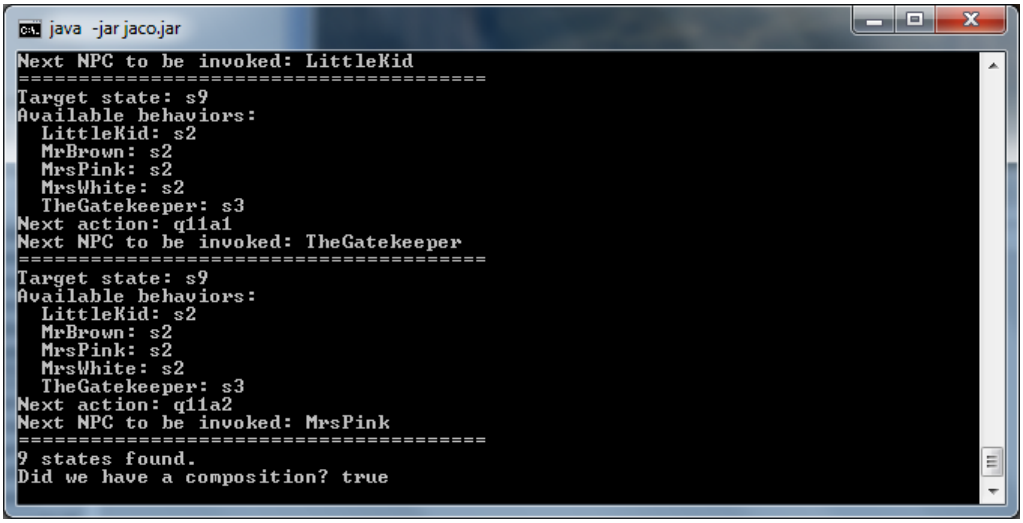


Figure 4.6: JaCO running locally.

In case the developer is interested in running JaCO locally, he/she would need the `.jar` package of the JaCO standalone server (downloadable from [6]) and run it from the command line.

In order to give to the reader an idea of how this communication is handled via code, here is the *method* from `CompositionScript.cs` that takes care of sending all the NPC Behaviors to JaCO via POST requests, using `WebRequest` from **.NET**:

[6] `http://jaco.dis.uniroma1.it/`

Listing 4.10: Code snippet from `CompositionScript.cs`

```
1  void sendBehaviors(){
2    //send the behavior of each NPC
3    foreach (string NPC in NPCList){
4      //compose the file path
5      string filePath = assetPath + "/GameXMLs/Behaviors/" + NPC + ".xml"
           ;
6      if(System.IO.File.Exists(filePath)){
7        string text = string.Empty;
8        using (StreamReader reader = new StreamReader(filePath, Encoding.
             UTF8))
9        {
10         text = reader.ReadToEnd();
11         Debug.Log(text);
12         reader.Close();
13       }
14       //create the POST request
15       WebRequest request =
16         WebRequest.Create(host + auth + "/behaviors");
17       request.Method = "POST";
18       request.ContentType = "text/xml; encoding='utf-8'";
19       byte[] byteArray = Encoding.UTF8.GetBytes(text);
20       Stream dataStream = request.GetRequestStream();
21       //write the data to the request stream.
22       dataStream.Write(byteArray, 0, byteArray.Length);
23       //cleanup the Stream object
24       dataStream.Close();
25       //get the response
26       WebResponse response = request.GetResponse();
27       //print the status
28       Debug.Log(((HttpWebResponse)response).StatusDescription);
29       //cleanup the response
30       response.Close();
31     }
32   }
33 }
```

Now, the reader may find confusing that we have been talking about FSMs in two different contexts: one mostly related to the player motion (see Fig-

ure 4.2) and the latter related to the interaction player-NPC. The two can be seen as one two-level FSM, where the first represents the higher level of the NPC logic, and the second a lower level for the *TALK* state.

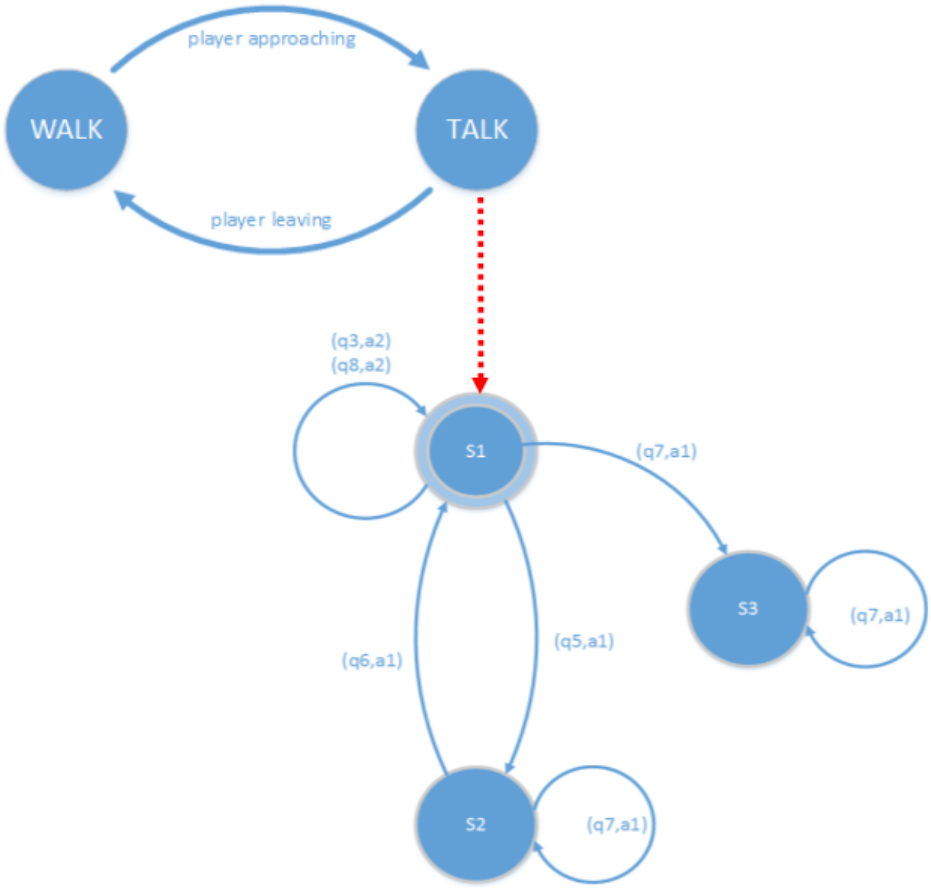


Figure 4.7: Mrs. White two-level FSM

4.3 Source Code

For brevity constraints, the source code will not be listed at the end of the document. Nonetheless, if the reader is interested in taking a deeper look at the System components and the related source code, the complete project is hosted on Bitbucket [7].

[7] `https://bitbucket.org/zhed/project-second-dawn`

Chapter 5

A demo unfolding

In order to give to the reader a better understanding of how this Dialogue System works, we'll now guide him/her through a possible demo unfolding, showing how different approachs lead to different consequences.
For completeness, and to show how such narrative evolves consistently, at the end of this commented example run of the developed mini-game we'll include the full script that unfolds from the two different player approaches we are about to review.

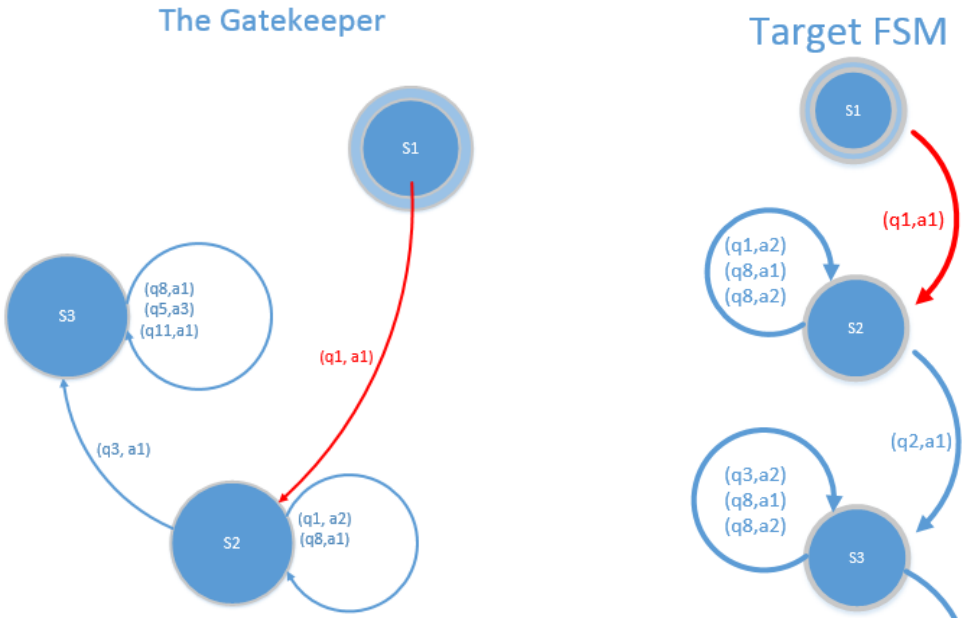
In order to make easier for the user to keep track of the game progress, a simple debug script has been written to show on screen the current game context, i.e. a list of all the states the *Behaviors* (both NPCs and Target) are currently in.

5.1 Two player approaches

All the Behaviors start out at their individual initial state, which is named *S1*. As depicted in Figure 3.4, at the start of the game no interaction is allowed apart from *(q1,a1)*, which is facilitated by the *Gatekeeper* and serves as the introductory *quest*, which introduces the player to the story. Selecting this interaction will take both the NPC and the Target to their corresponding state *S2*, at which point, the player can start interacting with other non-

player characters.

Figure 5.1: *(q1,a1)*, the first interaction with the *Gatekeeper*



(a) The Gatekeeper (b) Little kid

Figure 5.2: The Gatekeeper 5.2(a) and Target 5.2(b) FSMs

As shown in Figure 5.2(b), in this Target state, the player can be part of 4 different interactions: three of these are not critical for the progression of the story, and can be considered *filler quests* whose role is to increase the

credibility of the scenario. Furthermore, two of these 3 interactions consist of pairs of question and answers whose question is the same. Realistically, this happens when one is interested in having different points of view from different NPCs, according to the way they have been characterized.

What happens in states *S2*, *S3* and *S4* is basically the same: we still have a set of interactions that the player can take part into, but, ignoring how the NPC Behavior evolves, the Target state wil progress to the next state only if we succeed to trigger that particular story-critical interaction: in *S2*, for example, we can walk around and ask people about the neighborhood (*(q8,a#)*), or say hi again to the *Gatekeeper*, who this time will look upset (*(q1,a2)*), but the story will progress only when we approach the *Little Kid*, who knows something about the crime (*(q2,a1)*).

When the game will get interesting is in state *S5*, where the game features a story branching. Here the player can, in fact, choose to talk to *Mrs White* about *Mrs Pink* (*(q7,a1)*), or talk to *Mr Brown* (*(q9,a2)*), who will make the player suspicious about *Mrs White*, which is innocent, and therefore will take the player to the wrong track.

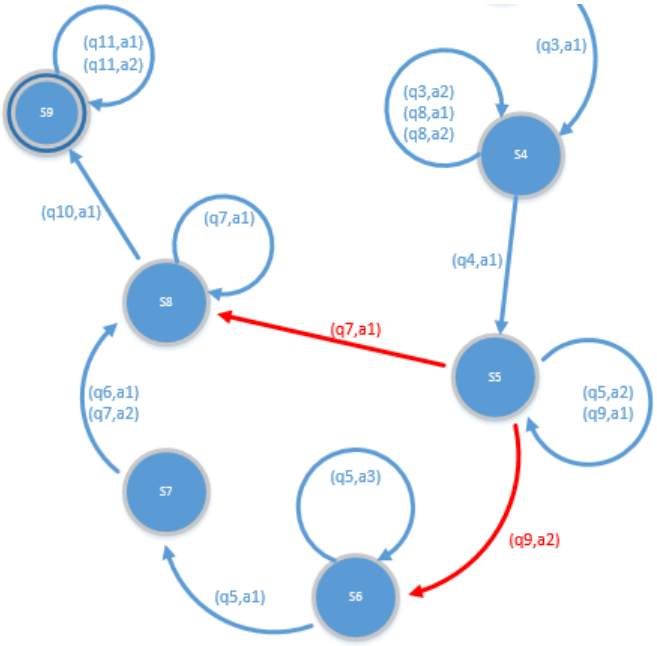


Figure 5.3: The story branching

The branches will eventually take to the same state, *S8*, where the player, having collected enough proofs against the true guilty, can finally accuse the criminal (((*q10,a1*)) and complete the game.

Figure 5.4: Endgame

We will now present the unfolding scripts from two different player approaches inspired to the described game walkthrough.

5.1.1 Unfolding script: first approach

(Q1,A1)

Player: *Hey there, what's going on?*

The Gatekeeper: *Finally! There is a terrible thing that just happened here! Go inside and investigate! Quick!*

(Q8,A2)

Player: *This place...do things like these happen often here?*

Mr. Brown: *Not at all, I wouldnt say so.*

(Q2,A1)

Player: *You, little kid: do you know anything about this crime?*

Little kid: *I could tell you...if only you could give me something in return...first.*

(Q3,A2)

Player: *The little kid is looking for something hes lost. Do you know what that is?*

Mrs. White: *Hmm, I have no idea. I barely see that kid around.*

(Q3,A1)

Player: *The little kid is looking for something hes lost. Do you know what that is?*

The Gatekeeper: *Oh, I guess I do! I found this in the yard, this morning.*

(Q4,A1)

Player: *Here you are. Can you tell me now?*

Little kid: *If I were you, I would ask Mrs. White over there...*

(Q7,A1)

Player: *What can you tell me about the woman in the pink dress?*

Mrs. White: *I saw her running away as I heard that scream...luckily Mr. Brown stopped her.*

(Q10,A1)

Player: *It was you! I asked witnesses! CONFESS!!!*

Mrs. Pink: *Oh no! Yes, it was me...but I swear, that was just a cookie!*

(Q11,A1)

Player: **proud air, whistling**

Little kid: *Good job, detective!*

5.1.2 Unfolding script: second approach

(Q1,A1)

Player: *Hey there, what's going on?*

The Gatekeeper: *Finally! There is a terrible thing that just happened here! Go inside and investigate! Quick!*

(Q8,A2)

Player: *This place...do things like these happen often here?*

Mrs. White: *Not at all, I wouldnt say so.*

(Q1,A2)

Player: *Hey there, what's going on?*

The Gatekeeper: *Are you still here?!? Run inside! The criminal might well still be around!*

(Q2,A1)

Player: *You, little kid: do you know anything about this crime?*

Little kid: *I could tell you...if only you could give me something in return...first.*

(Q3,A1)

Player: *The little kid is looking for something hes lost. Do you know what that is?*

The Gatekeeper: *Oh, I guess I do! I found this in the yard, this morning.*

(Q4,A1)

Player: *Here you are. Can you tell me now?*

Little kid: *If I were you, I would ask Mrs. White over there...*

(Q9,A2)

Player: *Mrs. White looks overly nervous. What can you tell me about her?*

Mr. Brown: *I saw her from behind the window, there in the kitchen, just before it turned into the crime scene.*

(Q5,A1)

Player: *Confess! Its you the one who committed the crime!*

Mrs. White: *I dont know what youre talking about! I dont even like cookies!*

(Q6,A1)

Player: *Sorry! It might well not be you after all...*

Mrs. White: *Obviously not! Youre accusing the wrong person! Mrs. Pink, SHE was acting suspiciously!*

(Q10,A1)

Player: *It was you! I asked witnesses! CONFESS!!!*

Mrs. Pink: *Oh no! Yes, it was me...but I swear, that was just a cookie!*

(Q11,A2)

Player: **proud air, whistling**

Mrs. Pink: *Im not gonna go to jail, am I?*

Chapter 6

Conclusions and future work

In these chapters we've first introducted the motivation behind the development of our Dialogue System, comparing such solution with different existing approaches towards the implementation of dialogues in videogames. In later chapters, we've reviewed some literature related to *Interactive Storytelling*, which is the research field this work belongs to, and Behavior Composition, which our work is based on. Then, we've described in detail the different components the Dialogue System is composed of and finally we've showed how a videogame can be developed making use it, analyzing the mini-game developed for this purpose.

Hopefully, at this point, the reader can sense the flexibility of such system and the benefits of taking advantage of Behavior Composition in Videogame Development. As more and more videogames feature open-world scenarios and the number of NPCs in the scene drastically increases, a mantaiblable approach to manage dialogues and, more generally, player-AI interactions, is a huge benefit.
Furthermore, avoiding to hard-code the dialogue lines into the NPC logic can open up new tracks in terms of gameplay, where AI can react in a less predictable and more immersive way, and still being able to guarantee that the flow of the narration complies with a designed target process.

Nonetheless, this is a young approach, and we believe that at this stage this Dialogue System still presents assumptions that in next versions should be dropped in order for the solution to be possibly used in a broader variety of projects.

The following are possible directions for future work:

- Handle more complex dialogues, such as multi-line answers and clustering question and answer sequences into atomic interactions, when desired.
- Implement an interface for the game designer in order to author and store dialogues in a more intuitive way without dealing with the XML syntaxis. This can be a Unity add-on or an external tool.
- Provide dialogues with the possibility to include "consequences" fields that edit game world variables. These consequences can take care of alterations of the game world whose importance is not story-critic, such as opening doors after having received the corresponding key, or maybe they can be used to enrich the descriptive power of the Target FSM.
- Give a finer control to the game designer over the distribution of dialogues at runtime. He/she can, for example, formalize a set of criteria, according to which, for instance, assuming that the same interaction could be facilitated by multiple NPCs, interacting with a subset of these could be funnier, or just preferrable, than interacting with the rest of them. This way the Dialogue System could help to push the player towards the directions preferred by the designer.
- Provide an interface to easily fetch and review Behaviors loaded on JaCO in previous working sessions, delete them (and upload them again, if they were meant to be updated), as well as being able to load new ones. This can be useful, for example, when one decides to change one single Behavior and keep the others as they already are.
  This boils down, in practical terms, to store, if desiderd, the `client_id`

the *JaCO* web service returns to the user in order to be authenticated in the following requests.
Furthermore, in order to delete or update Behaviors on the server the interface should support, along with POST and GET, the DELETE HTTP request.
The reader may refer to [1] for a complete JaCO REST API review.

As a newer version of JaCO is currently in development and new ways to interact with such web service will be availabe, it will be interesting to study how the Dialogue System here presented could be possibly expanded.

[1] `http://jaco.dis.uniroma1.it/`

List of Figures

Bibliography

[1] J. Porteous, M. O. Cavazza, and F. Charles, "Applying planning to interactive storytelling: narrative control using state constraints," *ACM Transactions on Intelligent Systems and Technology*, 2010.

[2] M. O. Riedl and V. Bulitko, "Narrative control using state constraints," *AI Magazine*, 2013.

[3] M. Mateas and A. Stern, *Proceedings of Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2005.

[4] M. Kelso, P. Weyhrauch, and J. Bates, "Dramatic presence," *Presence: Teleoperators and Virtual Environments*, no. 2, 1993.

[5] G. De Giacomo, F. Patrizi, and S. Sardiña, "Automatic behavior composition synthesis," *Artif. Intell. 196*, pp. 106–142, 2013.