An Open Source Architecture for Building Interactive Dramas

Vinícius M. Müller
DGiovanni Project (http://dgiovanni.sourceforge.net/)

Abstract

This work presents *DGiovanni*, an open source multi-agent architecture for building interactive dramas. The architecture has been developed in JAVA and uses the Jason's BDI engine, being the Jason's agent-oriented programming language utilized as the means for performing the drama management and for authoring the characters' behaviors. Additionally, it makes use of ontologies to support the creation of different stories and to feed the system with story-related information. Also, the architecture can help in the development and research of interactive dramas, by including several facilities for developing the story. Finally, in order to demonstrate the use of the architecture, it has been also created a story implementation that supports some interactivity mechanisms such as a simple Natural Language Processing.

Keywords:: Interactive Drama, Multi-agent systems, Ontologies

Author's Contact:

vmmuller@users.sourceforge.net

1 Introduction

A "... *story* is a narrative of events arranged in time sequence" [Forster 1956, p.30, emphasis added]. A *play*, on the other hand, "... tells a certain kind of story": "... one that imitates an action" [Rush 2005, p.30]. Certainly, in this case, action means more than having someone running or doing something exciting. Actually, there are plays, for example, where characters just speak with each other [Rush 2005]. Thus, this action must have "... a certain kind of depth" [Rush 2005, p.23]. In this sense, the "... unique kind of action ..." that can be found in a drama is the *dramatic action* [Rush 2005, p.30]. "A dramatic action is a specific event that occurs over a limited time in which a significant change occurs" [Rush 2005, p.23]. "Dramatic action depends on *conflict*: If there is no conflict, there is no drama" [Pritner and Walters 2005, p.53, emphasis added]. Hence, a dramatic event demands a conflict between characters with "... mutually opposed wants, desires, or needs" [Pritner and Walters 2005, p.53]. In [Forster 1956] view, like story, a plot is a narrative of events, but here the emphasis on *causality* is stressed. " 'The king died and then the queen' is a story. 'The king died and then the queen died of grief' is a plot" [Forster 1956, p.86]. Indeed, "... plot refers to the deliberate selection and arrangement of the incidents ..." [Rush 2005, p.35]. In this regard, according to [Rush 2005], there are several ways a play may be organized. The author can arrange the incidents in the same order they happen in the story; that is, in a chronological fashion: *linear order*. Conversely, a story may also be not chronologically organized: *nonlinear order*. The figure 1 shows the classical *well-made play* structure where events occur in a chronological sequence and are causally connected.

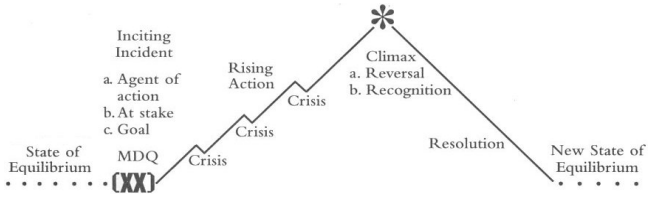
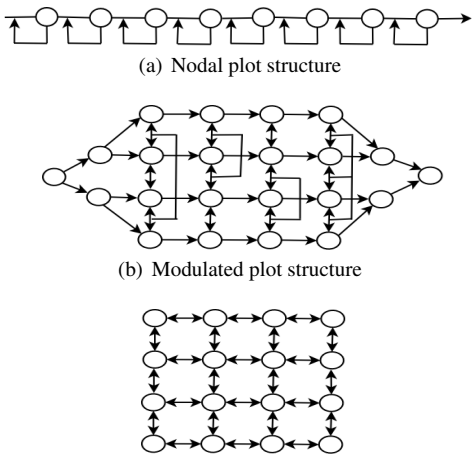


Figure 1: *Key points of a typical well-made play*
Reference: [Rush 2005, p.39]

Knowing that plot relates to the *process* [Tobias 1993] of selecting and arranging the incidents through time [Rush 2005; Mckee 1997], one question must still be answered: how the addition of interactivity may affect the plot? In this context, [Meadows 2002, p.238] defines an *interactive narrative* as "... a form of narrative that allows someone other than the author to affect, choose, or change the events of the plot". Here, plot is still a function of time, but now the act of choosing when the events occur is responsibility of both the author and the player [Meadows 2002]. [Meadows 2002] also describes two primary forms of interactive narrative: the *impositional* and the *expressive*. In the first, the plot is strongly determined by the author that guides the player with strict sets of rules, thus allowing him a narrow margin of choices and just a few moments of interactivity. Conversely, in the second, it is the player that heavily influences the plot, being allowed to walk freely, explore, and change the environment. In addition, [Meadows 2002] points out that seeking the appropriate balance between these two forms is a major challenge that must be faced. The figure 2 shows examples of three types of plots that represent the impositional and the expressive interactive plot structures[1].



(a) Nodal plot structure

(b) Modulated plot structure

(c) Open plot structure

Figure 2: *Interactive narrative plot structures*
Reference: [Meadows 2002, pp. 64-66]

The *nodal* plot structure, figure 2(a), is the one where there is "... a series of noninteractive events, interrupted by points of interactivity" [Meadows 2002, p.64]. This plot structure is the most impositional and has the most support for the classic dramatic arc. This structure has one beginning and usually two endings (one where the player dies and other where he finishes the game) [Meadows 2002]. In addition, the plot direction may not be changed by the player, which may only influence the plot progression speed in the course of its linear path [Hammond et al. 2007]. The *modulated* plot structure, figure 2(b), has less support for the classic dramatic arc, and does not necessarily determines the order of the events to be followed [Meadows 2002]. Here,

> "player action chooses which path the plot will follow by choosing from finite sets of pre-defined options at fixed decision points in the plot ... These decision points provide affordances for player agency, but their finite nature means that agency is somewhat limited" [Hammond et al. 2007, p.388].

Lastly, the *open* plot structure, figure 2(c), is the one where the focus lies on the exploration, modification and investment, being the dramatic arc usually abandoned. Moreover, the story is often characterized by the development of characters or environments [Meadows 2002]. "This form of narrative has no specific starting point in the sense that there is an event that begins the story"[Meadows 2002, p.66].

[1] Differently from that presented in figure 1, these "... don't have much to do with emotional punch or aesthetic interest" [Meadows 2002, p.63]

According to [Ryan 2008], the first step in designing an interactive narrative is to choose the type of the story. In this case, [Ryan 2008, p.10] believes that the implementation of the dramatic plot is the most difficult because of "... its emphasis on the evolution of interpersonal relations". This kind of interactive story that is characterized by the dramatic plot is called *interactive drama* and may be defined as follows:

> "Interactive Drama is a narrative genre on computer where the user is one main character in the story and the other characters and events are automated through a program written by an author. Being a character implies choosing all narrative actions for this character" [Szilas 2005, p.193].

In other words, the main idea of an interactive drama consists of enabling the player to be a major character within the story, interact with the computer-generated characters, and have a strong influence in the course of events [Szilas et al. 2008].

In this context, this work presents an open source multi-agent architecture for building interactive dramas. It has been mainly inspired by the interactive drama Façade[2] because this is 'fully-realized' and released as freeware for download. However, differently from Façade, for the developed architecture, none specific-purpose authoring language has been created. Actually, for performing the drama management and the authoring of the characters' behaviors, the well-known agent-oriented programming language Jason[3] is utilized. In addition, the developed architecture also uses ontologies to support the creation of different stories and to feed the system with story-related information. Finally, in order to demonstrate the use of the architecture, it has been also created a story implementation that supports some interactivity mechanisms such as a simple Natural Language Processing. Notice, however, that the use of sophisticated computer graphics and highly interactive mechanisms is beyond the scope of this experiment.

The text is organized as follows: the section 2 shows some related works and describes the interactive drama Façade. The section 3 presents an overview of the developed architecture. An experiment in building an interactive drama using the developed architecture is shown in section 4. Finally, the section 5 presents the final considerations and discusses about future works.

2 Related Work

Interactive Drama has attracted a growing interest over the years and several approaches to the problem have already been presented: project Oz architecture [Loyall and Bates 1991]; IDtension [Szilas et al. 2008], IDA [Magerko 2002], Façade [Mateas and Stern 2003].

This work has been mainly inspired by Façade because that is a fully-realized one-act interactive drama and is released as freeware for download [Mateas and Stern 2003]. "The Façade architecture integrates story level interaction (drama management), believable agents, and shallow natural language processing in the context of a first-person, graphical, real time interactive drama" [Mateas and Stern 2003, p.7]. Façade follows the *Neo-Aristotelian* theory of interactive drama, which is based on Aristotle's dramatic theory, but with some modifications to address the interactivity added by player agency [Mateas 2002]. In the Neo-Aristotelian approach, to deal with the tension existing between interactive freedom and story structure, the "... player has been added to the model as a character who can choose his or her own actions ... But this ability to take action is not completely free ..." [Mateas 2002, p.24]. He is constrained by what is possible for him to do in the representation (*material cause*), and even not perceiving it directly, by what the story is trying to be (*formal cause*) [Mateas 2002].

Façade is composed of several threads running in parallel: A real-time rendered 3D story world, the agent Trip, the agent Grace, the player's avatar agent, a thread for handling the natural language processing, and a drama manager [Mateas and Stern 2003]. The *believable agents* Trip and Grace are endowed with the ability to perform several intelligent activities such as to gaze, speak, walk, etc. They consist of a large collection of behaviors written in ABL (A behavior language) [Mateas and Stern 2003]. Based on the Oz project architecture Hap [Loyall and Bates 1991], ABL is a *reactive planning language* in which an activity (e.g. walking to the player) is represented as a goal, and each goal may be achieved via one of the supplied behaviors [Mateas and Stern 2003].

> "A *behavior* is a series of steps, which can occur sequentially or in parallel. Typically, once a behavior completes all of its steps, it succeeds and goes away. However, if any of its steps fail, then the behavior itself fails and the goal attempts to find a different behavior to accomplish its task, failing if no such alternative behavior can be found" [Mateas and Stern 2003, p.8-9, emphasis added].

These steps "... can be subgoals, mental acts (bits of computation, often used to update character memory), or primitive acts (actions, such as performing an arm gesture, that are native to the game world)" [Zang et al. 2007, pp. 2-3]. The behaviors are dynamically picked out to achieve goals. Thus, distinct behaviors can be used for attaining the same goal in different contexts. For instance, if a character has the goal of expressing anger, this can be reached through either a behavior in which he screams or other where he punches a hole in the wall [Zang et al. 2007]. ABL supports *sequential behaviors*, where the steps are accomplished in a serial order; *parallel behaviors*, where they are accomplished simultaneously; and *joint behaviors*, where there is a mechanism for realizing a multi-agent coordination [Mateas and Stern 2002].

In relation to the *drama management*, in the Façade's architecture, "... character behavior is organized around the dramatic beat" [Mateas and Stern 2002, p.3]. In dramatic theory, the *beat* is "... the smallest unit of dramatic action, consisting of a short dialog exchange or a small amount of physical action" [Mateas 2002, p. ii]; however, the Façade's beats ended up being larger than the canonical beats. The *drama manager* agent is the beat sequencer, which chooses the subsequent beat in the story by considering the preceding interaction history [Mateas and Stern 2003]. "As the story progresses, beats are sequenced in such a way as to be responsive to recent player interaction while providing story structure" [Mateas 2002, p.4]. A beat supplies the characters Grace and Trip with a collection of behaviors appropriate for a specific context [Mateas and Stern 2003]. Beat behaviors are created in ABL, and are classified as *beat goals*, *handlers* ("responsible for handling player interaction"), and *cross-beat behaviors* ("behaviors that cross beat goal and handler boundaries") [Mateas and Stern 2002, pp. 3-4].

> "In a beat sequencing language the author annotates each beat with selection knowledge consisting of preconditions, weights, weight tests, priorities, priority tests, and story value effects — the overall tension level, in Façade's ... The unused beat whose preconditions are satisfied and whose story tension effects most closely match the near-term trajectory of an author-specified story tension arc (in Façade, an Aristotelian tension arc) is the one chosen; weights and priorities also influence the decision" [Mateas and Stern 2005, p.4].

The player agent does not take part in the story world; however, it is responsible for sensing the player's actions and delivering this information to the other agents. The player agent is also written in ABL [Mateas and Stern 2003]. "Player interaction alters the performance of a beat (*local agency*), and can have longer term effects on future beats (*global agency*)" [Mateas and Stern 2003, p.11, emphasis added]. The player types to speak to the characters, whereas they speak loudly their own dialog. To handle player interaction, the natural language processing thread tries to interpret the player's action, and map it into one or more discourse acts [Mateas and Stern 2003]. "A *discourse act* is a concise representation of the general meaning of the player's action" [Mateas and Stern 2003, pp. 11-12, emphasis added]. Examples of discourse acts used in Façade are 'agree', 'disagree', 'positive exclaim', 'express happy', etc. Subsequently, considering the current beat, the discourse act is mapped into a potential reaction [Mateas and Stern 2003].

[2] http://www.interactivestory.net/

[3] http://jason.sourceforge.net/

3 An architecture for building interactive dramas

This section presents the *DGiovanni* multi-agent architecture for building interactive dramas. It has been developed in JAVA[4] and uses the Jason's BDI engine. Jason is an interpreter that implements the operational semantics of the AgentSpeak language, and provides a platform for developing multi-agent systems, including many user-customizable features. "An *agent* is anything that can be viewed as perceiving its *environment* through *sensors* and acting upon that environment through *actuators*" [Russell and Norvig 2009, p.34, emphasis in the original]. In this case, a "*percept* is an item of information received from the environment by some sensor" [Padgham and Winikoff 2004, p.8, emphasis in the original]. An *action*, on the other hand, "... is basically an agent's ability to affect its environment" [Padgham and Winikoff 2004, p.8].

AgentSpeak is inspired on the *belief-desire-intention* (BDI) model, which is a model where computer programs may be thought as they have a "mental state", in the sense that they have computational analogues of *beliefs*, *desires* and *intentions* [Bordini et al. 2007, p.15]. The *beliefs* are the information that the agent has about the environment, even if this information is inaccurate or out of date; *desires* are the world condition that the agent might like to achieve [Bordini et al. 2007]; and *intentions* are related to the course of action currently chosen [Padgham and Winikoff 2004]. Based on its beliefs, desires, and intentions, a BDI agent determines what to do by means of the decision-making *practical reasoning* model. This model consists of two different activities: deliberation (decision of what is to be achieved, i.e. intentions) and planning (decision of how to act to accomplish the expected state of affairs, using the available means) [Bordini et al. 2007]. In relation to planning, the approach that has been adopted in Jason is that the programmer develops collections of partial plans offline (i.e. at design time). In this case, to handle any goal the agent is currently working towards, the agent is responsible for assembling plans for execution at run time [Bordini et al. 2007]. The Jason interpreter runs an agent program operating by means of a *reasoning cycle* which consists of the following steps [Bordini et al. 2007, pp. 67-86]:

1. Perceiving the environment;
2. Updating the belief base;
3. Receiving communication from other agents;
4. Selecting 'socially acceptable' messages;
5. Selecting an event;
6. Retrieving all relevant plans;
7. Determining the applicable plans;
8. Selecting one applicable plan;
9. Selecting an intention for further execution;
10. Executing one step of an intention.

This work has taken a multi-agent approach to interactive drama because of several reasons. One is the reduction of coupling that results from the autonomy, robustness, reactiveness, and proactiveness of agents [Padgham and Winikoff 2004]. In addition to reducing coupling, agents can be used in unpredictable, unreliable, and dynamic environments, in which fails may occur and recovery must be done. Moreover, the proactiveness and the reactiveness of the agents make them more human-like in the manner they deal with problems: a fact that makes them useful in applications such as games, in which software agents substitute humans [Padgham and Winikoff 2004]. Furthermore, according to [Norling and Sonenberg 2004], the use of the BDI paradigm may be interesting to the creation of characters that must exhibit a wide range of complex behaviors and interact with players that are usually unpredictable.

In summary, the *DGiovanni* architecture has the following features:

- Implementation of the infrastructure for communication and control of the agents creation, running, and destruction;
- Customizable framework, compounded of classes and interfaces that have a terminology related to drama (i.e. including terms like *StoryCharacter*, *StorySetting*, etc.);
- Support for the creation of different stories via the utilization of ontologies that are also used to feed the system with story-related information;
- An interface focusing on the research of interactive dramas:
  - Visualization of the dramatic arc, selected beats/behaviors history and beat markers to delimit the beats (figure 3), state of the agents and environment (figure 4), and analysis of the player's input (figure 5);
  - A new tab for each restart of the story;
  - Support for showing text and emoticons;
  - Generation of output in HTML format.
- Facilities for playing MP3 sounds (using JLayer[5]);
- A story implementation that uses some interactivity mechanisms such as a simple natural language processing.

This work has adopted several ideas used in the interactive drama Façade: the use of *discourse acts* [Mateas and Stern 2003]; the use of the *beat* as the central building block of the interactive story [Mateas and Stern 2000]; the use of the term *behavior* to mean an activity that a character may perform [Zang et al. 2007]; the use of a *drama manager* to provide the high-level plot decisions [Mateas and Stern 2005]. However, differently from Façade, for the developed architecture, none specific-purpose authoring language (such as the ABL and the beat sequencing language) has been created. Actually, for performing the drama management and the authoring of the characters' behaviors, the Jason is utilized. Jason is used for some reasons: First, the Façade's authoring tools are not publicly available at this time. Second, Jason is implemented in JAVA, thus multi-platform. Third, it is available open source. Fourth, besides being very powerful, it is highly customizable. Fifth, the elegance of the AgentSpeak notation is particularly appealing to the symbolic representation of beats and behaviors. Last, Jason is also a reactive planning language [Bordini 2010]. [Mateas 2002, p.105] comments about the importance of the use of a reactive planning language in the interactive drama Façade:

> "One of the strengths of a reactive planning language such as ABL is the possibility of rich, dynamic combinations of behavior. While a behavior author defines distinct behaviors for specific situations, much of the richness of an ABL character comes from the interactions between multiple behaviors as the character pursues multiple goals in response to changes in the environment".

The developed architecture itself is shown in figure 6. The drama manager controls the characters by sending messages to them, and they reply with a message after completing the requested behavior. Both the drama manager and the characters can affect (via an *action*) and sense (i.e. *perceive*) the environment. As a direct consequence of being the implementation of Jason agents, the characters (including the player) can be seen as divided in three parts:

- A kind of *mind* that has a belief base, a plan library, and implements the default selection functions of the AgentSpeak. It uses the asl file that defines the behaviors of the characters.
- The base architecture class that defines the overall agent architecture, being a kind of *body*. During the reasoning cycle of the agent, its methods are called to take actions, establish communication, and get the perceptions.
- The implementation that is visible in the story world and that provides the concrete perception, action, and communication.

The player character is defined by the same class as the non-player characters (NPCs); however, here the user's implementation must supply a module for handling the player's input. In this case, the architecture provides classes that abstract *expressions* and *environment affecting actions*. The first, which are accomplished by sending a Jason message, are used to communicate feelings, opinions, etc. by means of words or actions. The second relates to the actions that are executed by an entity in order to affect the environment (e.g. knock on the door). In such cases, percepts are added to the environment after the execution of the actions.

[4] http://www.oracle.com/technetwork/java/index.html

[5] JLayer MP3 library (http://www.javazoom.net/javalayer/javalayer.html)

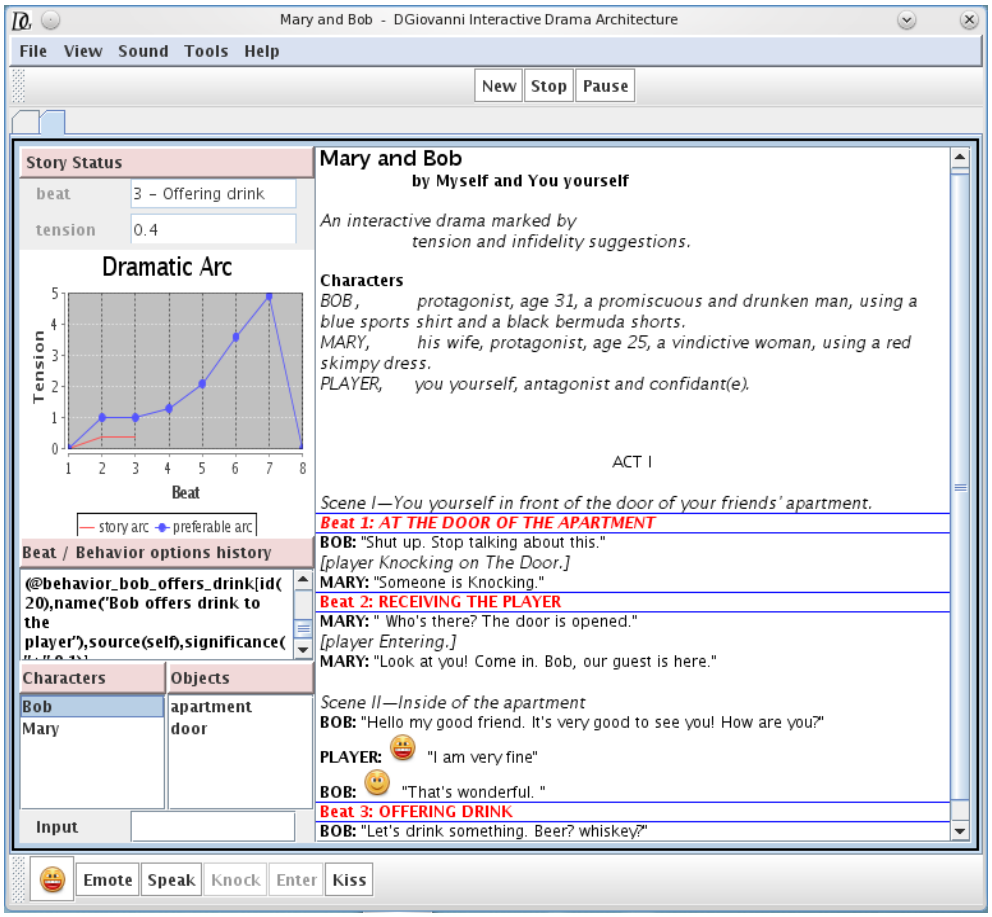


Figure 3: *The architecture's default interface: enabling beat markers*
Emoticons from the Pidgin emoticon theme (http://www.pidgin.im/). Dramatic arc created with JFreeChart (http://www.jfree.org/jfreechart/)

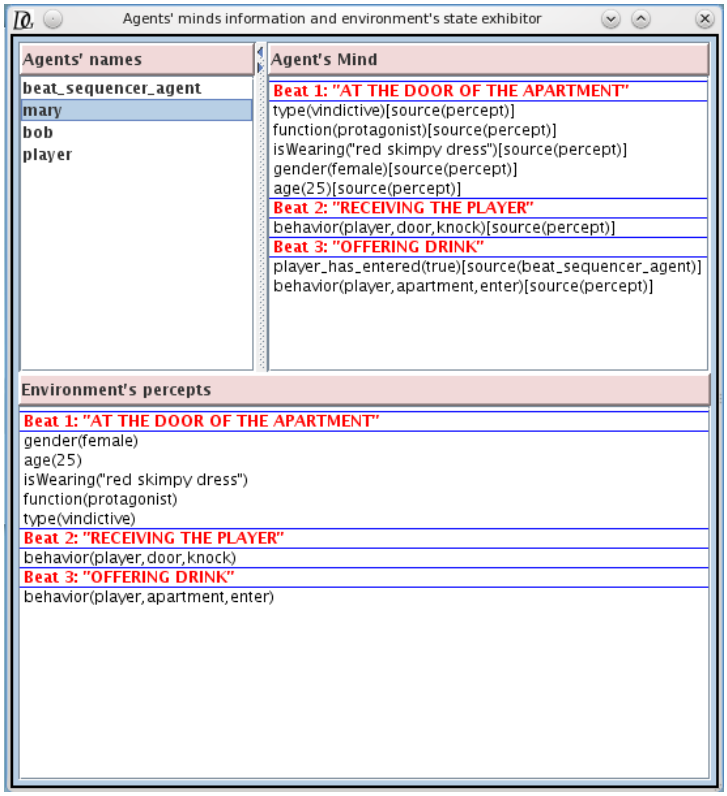


Figure 4: *State of the agents and environment*

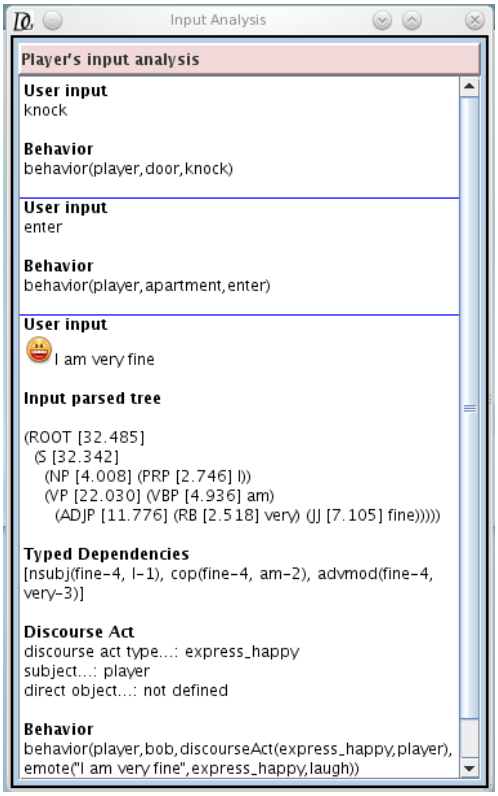
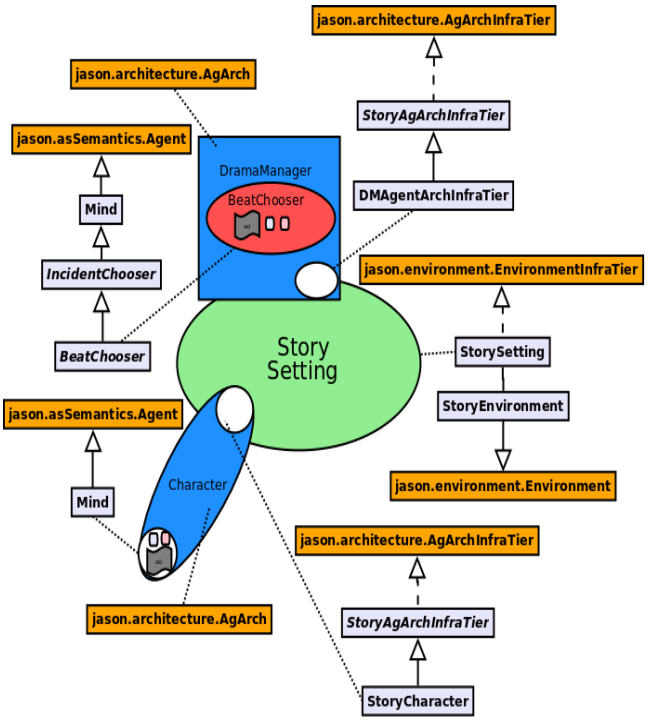
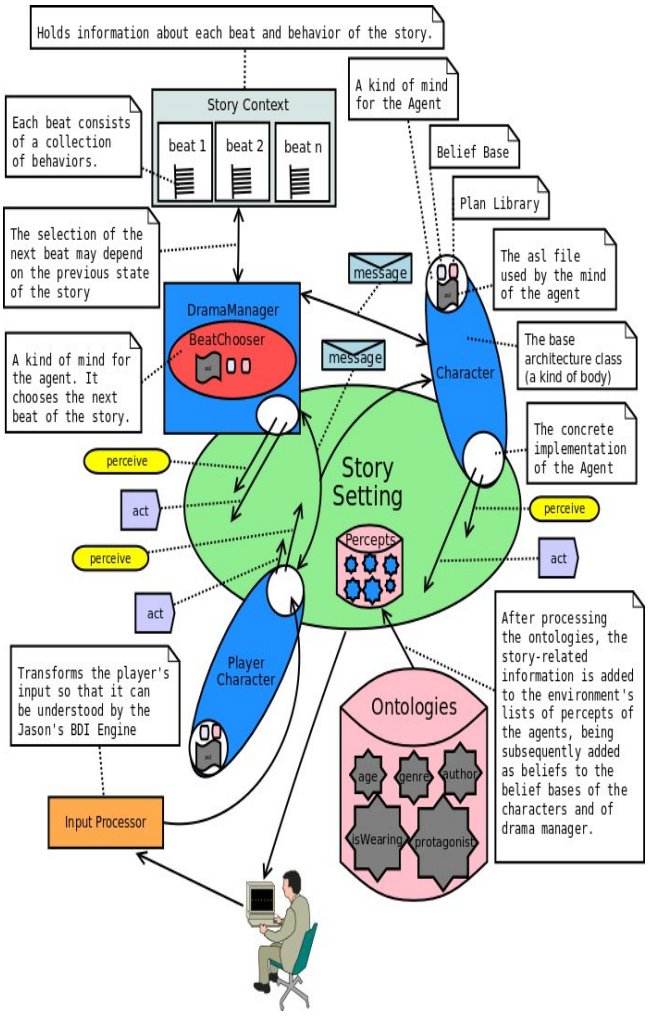


Figure 5: *Analysis of the player's input*



(a) The implementation of the infrastructure for communication and control of the agents



(b) The developed architecture's operation

Figure 6: *The developed architecture*

The drama manager agent is similar to the other agents. Nevertheless, there are some differences. The main difference is that the class *jason.asSemantics.Agent* is extended to provide the drama management; that is, the *BeatChooser* is used to select the next beat of the story according to some criteria defined by the user of the architecture, being the selection of the next beat done in the eighth step of the Jason reasoning cycle (i.e. *selecting one applicable plan*). For this, the architecture also provides classes for holding information about each beat and behavior of the story so that in the selection of the next beat may be considered the story context. The asl file used by the *BeatChooser* has other purpose as well. It contains the main structure of the story; i.e. the definition of the beats. Indeed, in this architecture, it is considered a beat every plan with the word *beat* as functor[6] and a behavior every plan with the word *behavior* as functor. Lastly, other difference between the drama manager agent and the characters is the fact that the concrete implementation of the drama manager (i.e. *DMAgentArchInfraTier*) is not visible to the player in the story world.

DGiovanni also makes use of ontologies to support the creation of different stories and to feed the system with story-related information. An *ontology*, according to [Noy and McGuinness , p.3, emphasis in the original],

> " ... is a formal explicit description of concepts in a domain of discourse (*classes* (sometimes called *concepts*)), properties of each concept describing various features and attributes of the concept (*slots* (sometimes called *roles* or *properties*)), and restrictions on slots (*facets* (sometimes called *role restrictions*)). An ontology together with a set of individual instances of classes constitutes a *knowledge base*".

In the developed architecture the ontological knowledge is represented by the Web Ontology Language (OWL), more specifically the OWL DL sublanguage[7]. The architecture uses the OWL API[8] and the reasoner HermiT[9] for loading information defined in the architecture's knowledge base. The knowledge base hierarchy, figure 7, is divided into three ontologies: one tbox and two aboxes.

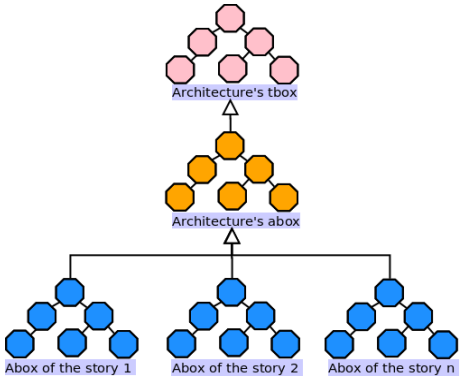


Figure 7: *Knowledge base hierarchy*

A "... t-box contains the axioms defining the classes and relations in an ontology, while the a-box contains the assertions about the individuals in the domain" [Jena 2011]. Thus, the tbox defines the concepts used by the architecture, whereas the architecture's abox contains information that is shared among all story implementations: e.g. name of authors; character functions, genres, etc. Hence, for the implementation of a story, the user of the architecture is responsible for the creation of an abox that imports the architecture's abox and for the assertion of the values of this new abox. For every story implementation, the user of the architecture should assert the values of the created abox according to the architecture's tbox presented in figure 8. For example, in the new abox, the user of the architecture may define the location of JAVA classes that are loaded through reflection, the location of the asl files that contain

[6] For example, *beat(receiving_the_player, HAS_KNOCKED)*
[7] http://www.w3.org/TR/owl-guide/
[8] http://owlapi.sourceforge.net/
[9] http://hermit-reasoner.com/

the characters' behaviors, the location of the asl file where the beats are coded, and information that is added automatically to the environment's lists of percepts (e.g. information about a scene, the age of the characters, etc.). In fact, after processing the ontologies, the story-related information is added to the environment's lists of percepts of the agents, being subsequently perceived and added as beliefs to the belief bases of the characters and of drama manager.

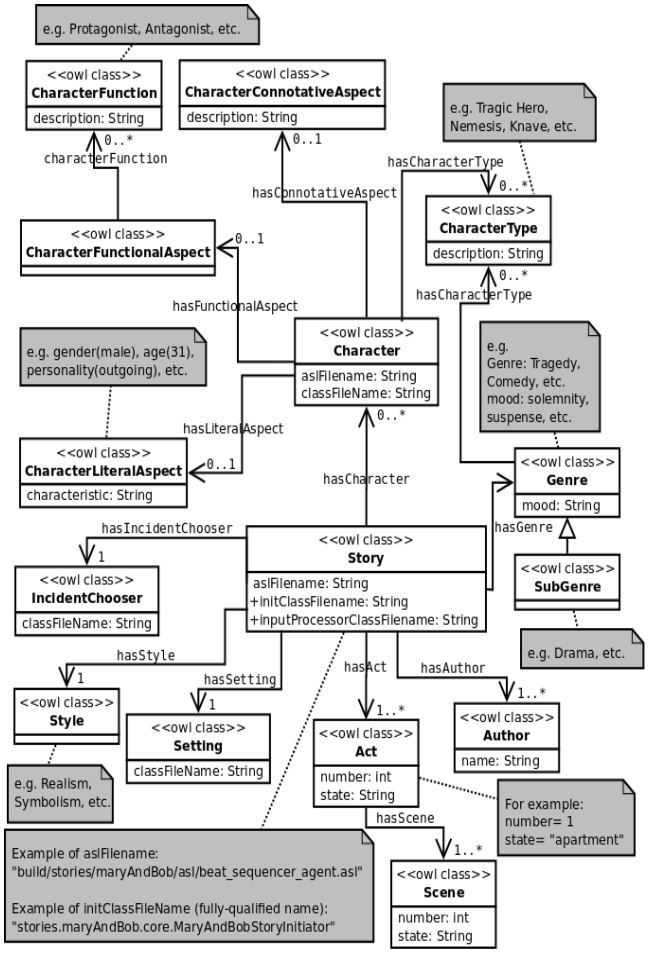


Figure 8: *The tbox of the architecture*

4 An experiment in building an interactive drama

You are invited to go to your friends' apartment: Mary and Bob, a couple having serious marriage problems. Bob, a man marked by promiscuity and alcoholism. Mary, a charming and vindictive woman, determined to get even. Can you overcome the temptation?

In order to demonstrate the use of the architecture, together with *DGiovanni*, it is included a prototype of an interactive drama called *Mary and Bob*. According to Pressman [Pressman 2001], a *prototype* is "... a model of the software to be built" [Pressman 2001, p.289] and "... can serve as the 'first system' " [Pressman 2001, p.31]; that is, *prototyping* may offer the best approach whenever the developer is not certain of how the human/machine interaction should happen, and when it is not identified detailed input, processing, or output requirements [Pressman 2001]. Indeed,

> "... any application that creates dynamic visual displays, interacts heavily with a user, or demands algorithms or combinatorial processing that must be developed in an evolutionary fashion is a candidate for prototyping" [Pressman 2001, p.289].

In this case, the selected prototyping approach is the *evolutionary prototyping* where the "... prototype of the software is the first evolution of the finished system" [Pressman 2001, p.289]. Therefore, despite the fact that the use of sophisticated computer graphics and highly interactive mechanisms is beyond the scope of this experiment, there is a possibility of further improvements in the future.

Mary and Bob is an one-act story (consisting of two scenes) that focuses on the intensification of a conjugal *conflict*. The story's structure is similar to the *well-made play* presented in figure 1. Nevertheless, the *inciting incident* (the cheat) occurs before the start of the story. In the case of the player's being curious, he can eavesdrop at the apartment's door and listen about the cheat. As soon as the player enters his friends' apartment, he starts to face several *crises* caused by their enticing suggestions. The story moves this way until the *climax*, when the player is forced to take his ultimate decision. Mary and Bob are the two *protagonists* of the story, being Bob interested in convincing the player to drink and have fun, and Mary seeking to revenge on Bob for the cheat by trying to seduce the player. On the other hand, the player is both an *antagonist* and a *confidant*. Antagonist because he may block the characters from achieving their desires, and confidant in the sense that he is the one for whom the NPCs deliver secret information. Three ends are possible: (1) The player is expelled by Bob, (2) the player is expelled by Mary, and (3) the player flees covered in blood. The beat has been used as the central building block of the story, consisting it of at most eight beats. The table 1 shows these beats together with the minimum and maximum tension that each beat can add to the story, plus the number of player behaviors supported in each beat.

Name	Minimum Tension	Maximum Tension	Player Behaviors
At the door of the apartment	+0.0	+1.2	2
Receiving the player	+0.0	+0.2	5
Offering drink	-0.1	+0.4	9
Bob suggests going to a massage parlor	+0.2	+0.9	4
Mary makes advances to the player	+0.3	+1.5	15
Making the player go away	+0.6	+1.3	0
Fleeing	-4.9	-4.9	0
Finis	+0.0	+0.0	0
—	—	—	more 6 misc behaviors

Table 1: *The beats of the Mary and Bob story*

The created story is somewhat *impositional*, allowing the player just a narrow margin of choices and a few moments of interactivity. Its plot structure is *nodal* (see figure 2(a)). Thus, there is a series of noninteractive events, interrupted by points of interactivity where the player can execute an action (e.g. knock on the door), or input a text or an emoticon (or both). He is constrained by what is possible for him to do in the representation (*material cause*), and by what the story is trying to be (*formal cause*). Finally, it has been used the architecture's default user interface (see figure 3) to show the story, and to provide the means for the player to interact with it. The figure 9 shows the abox of the story. This is used to feed the system with story-related information (e.g. what a character is wearing), and to define the location of JAVA classes that are loaded through reflection, the location of the asl files that contain the characters' behaviors and the location of the asl file where the beats are coded.

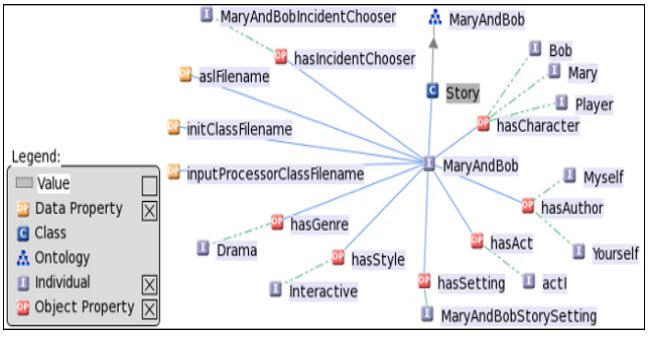


Figure 9: *The abox of the Mary and Bob story*
Image generated by NeOn toolkit (http://neon-toolkit.org/)

The story's structure is represented by the main structure asl file (i.e. the *beat_sequencer_agent.asl* file) whose fragment is presented in figure 10. After processing the ontologies, the story-related information is added to the environment's lists of percepts of the agents, being subsequently perceived and added as beliefs to the belief bases of the characters and of drama manager. The addition of the *act(1,apartment)* belief to the belief base of the drama manager triggers the *+act(1,apartment)* plan, and thus starts the story. As can also be seen in the figure 10, the information added to the belief bases of the agents (e.g. information about a scene, the age of the characters, etc.) is retrieved to be printed in the GUI.

```
+act(1,apartment)
 : true
 <- !presentation;
 dDialog.actDivision("I");
 ?act(1,scene(1,SCENE1_DIRECTION));
 ?act(1,scene(2,SCENE2_DIRECTION));
 dDialog.sceneDivision("I",SCENE1_DIRECTION);
 !beat(at_door).

+!presentation
  :true
<- ?author[AUTHOR1 | [AUTHOR2 | L ]];
  ?genre(GENRE_NAME,MOODVALUES);
  ?style(STYLE_NAME);
// Notice that the order of the loaded values may vary
  .nth(0,MOODVALUES,moodValue(MOOD_VALUEA));
  .nth(1,MOODVALUES,moodValue(MOOD_VALUEB));
  ?mood_description(MOOD_VALUEA,MOOD_VALUEB,MOOD_VALUE1,
                    MOOD_VALUE2);
  .send(bob,askOne,age(_),age(BOB_AGE));
  .send(bob,askOne,isWearing(_),isWearing(BOB_CLOTHES));
  .send(bob,askOne,function(_),function(BOB_FUNCTION));
  .send(bob,askAll,type(_),BOB_TYPES);
  //gets the nth term of the list
  .nth(0,BOB_TYPES,type(BOB_TYPE1));
  .nth(1,BOB_TYPES,type(BOB_TYPE2));
  .send(mary,askOne,age(_),age(MARY_AGE));
  .send(mary,askOne,isWearing(_),isWearing(MARY_CLOTHES));
  .send(mary,askOne,function(_),function(MARY_FUNCTION));
  .send(mary,askOne,type(_),type(MARY_TYPE));
  .send(player,askAll,function(_),PLAYER_FUNCTION);
  //gets the nth term of the list
  .nth(0,PLAYER_FUNCTION,function(PLAYER_F1));
  .nth(1,PLAYER_FUNCTION,function(PLAYER_F2));
  .concat("        ",BOB_FUNCTION,", age ",BOB_AGE,", a ",BOB_TYPE1," and ",
          BOB_TYPE2," man, using a ",BOB_CLOTHES,".",BOB_DESCRIPTION);
  .concat("      his wife, ",MARY_FUNCTION,", age ",MARY_AGE,", a ",MARY_TYPE,
          " woman, using a ",MARY_CLOTHES,".",MARY_DESCRIPTION);
  .concat("              by ", AUTHOR1," and ",AUTHOR2,AUTHORS);
  .concat("    you yourself, ",PLAYER_F1," and ",PLAYER_F2,".",
                              PLAYER_DESCRIPTION);
  .concat("An ",STYLE_NAME," ",GENRE_NAME," marked by",STORY_DESCRIPTION1);
  .concat("              ",MOOD_VALUE1," and ",MOOD_VALUE2," suggestions.",
                          STORY_DESCRIPTION2);
  dDialog.title("Mary and Bob");
  dDialog.println("");//adds a newline
  dDialog.println(AUTHORS,"bold");
  dDialog.println("");
  dDialog.println(STORY_DESCRIPTION1,"italic");
  dDialog.println(STORY_DESCRIPTION2,"italic");
  dDialog.println("");
  dDialog.println("Characters","bold");
  dDialog.present("bob ",BOB_DESCRIPTION);
  dDialog.present("mary",MARY_DESCRIPTION);
  dDialog.present("player",PLAYER_DESCRIPTION);
  dDialog.println("");
  dDialog.println("").
```

Figure 10: *Beginning the Mary and Bob story*

In *Mary and Bob*, it is considered a beat every plan with the word *beat* as functor and a behavior every plan with the word *behavior* as functor. The beats and the behaviors have *annotations* attached to themselves. Every beat has as annotations an *id*, a *name*, a *min_tension* (minimum tension), and a *max_tension* (maximum tension). On the other hand, each behavior has as annotations an *id*, a *name*, and a *significance* (the quantity of tension that the behavior adds to or subtracts from the beat). The beats are coded into the *beat_sequencer_agent* file. Inside this file every beat consists of a collection of behaviors; however, at run time each beat end up being the size of the canonical beat (i.e. a short dialog exchange or small amount of physical action). In relation to the drama management, just the fourth beat is defined explicitly via the drama manager implementation; the arrangement of the other beats is determined by the *beat_sequencer_agent* file used by this drama manager. In that case, if the story's tension is lesser than or equal to 1.3, the beat *Bob suggests going to a massage parlor* is added to the story.

The figure 11 shows a fragment of the beat *Receiving the player*. After the start of this beat (figure 11, step 1), the drama manager sends a message to the character Mary indicating that she must receive the player (figure 11, step 2). When this is done, the agent Mary sends a message to inform the drama manager that the action has been completed (figure 12, step 3). The drama manager receives the reply (figure 11, step 4) and this end up triggering the **+!behavior(bob_receive_the_player)** plan so that the drama manager sends a message to Bob ordering him to receive the player (figure 11, step 5). Bob welcomes the player and asks **"How are You?"** (figure 13, step 6). Next, for example, in the case of the player's replying with **"I am great today"**, it is added to the belief base of the drama manager the belief **behavior(player,bob,discourseAct(express_happy,player),"I am great today")**, which makes the drama manager send a message to the player agent asking him to print the player's input on the GUI (figure 11, step 7). As soon as the input has been printed, the player agent sends a message to the drama manager indicating that the action has been completed (figure 14, step 8). Now, the drama manager sends a message to the character for whom the player has replied (i.e. Mary or Bob), commanding a reaction to the player's utterance (figure 11, step 9)). Lastly, one character reacts to the player's utterance (figure 12 and 13, step 10).

```
// STARTING THE BEAT
@beat_receiving_the_player [id(2),
                            name("Receiving the player"),
                            min_tension("+",0.0),
                            max_tension("+",0.2)]                    1

+!beat(receiving_the_player, HAS_KNOCKED)
 :true
 <- !behavior(receive_the_player,HAS_KNOCKED).
// Mary receives the player.
// "Look at you! Come in. Bob, our guest is here. Bob, Our guest is here".
@behavior_mary_receives_the_player_that_has_knocked [ id(7),
                            name("Mary receives the player that has knocked"),
                            significance("+",0.0)]                   2
+!behavior(receive_the_player,HAS_KNOCKED)
 : HAS_KNOCKED == true
 <- .send(mary,achieve,receive_the_player(HAS_KNOCKED)).
// Mary has received the player that has knocked on the door.
+has_received_player(mary)                                            4
 :true
 <- !behavior(bob_receive_the_player).
//  Bob receives the player.
// "Hello my good friend. It's very good to see you! How are you?"
@behavior_bob_receives_the_player [id(8),
                            name("Bob receives the player"),
                            significance("+",0.0)]                   5
+!behavior(bob_receive_the_player)
 :true
 <- ?act1_scene2(SCENE2_DIRECTION);
   dDialog.sceneDivision("II",SCENE2_DIRECTION);
   .send(bob,achieve,receive_the_player).
// The player express happiness to the question "How are you?"
// "I am fine", "I am great","I am fine" and emote(HAPPY), etc.
@behavior_express_happy_with_subject[ id(16),
                            name("Player express happiness (with subject)"),
                            significance("-",0.1)]
+behavior(player,RECEIVER,discourseAct(DISCOURSE_ACT,player),Content)
  :context(receiving_the_player) & ( DISCOURSE_ACT==express_happy |      7
                            DISCOURSE_ACT==praise_comportment |
                            DISCOURSE_ACT==praise_aspect |
                            Content==emote(good))
 <- .send(player,achieve,happiness_expression(RECEIVER,Content));
 .abolish(behavior(player,RECEIVER,
                   discourseAct(DISCOURSE_ACT,player),Content)).
// Mary or Bob reacts with a happiness expression to the player's happiness.
@behavior_react_happiness_expression[ id(9),
                            name("Mary or Bob reacts to the player's happiness"),
                            significance("+",0.0)]                   9
+behavior(happiness_expression_realized(RECEIVER))[source(S)]
 : RECEIVER == mary | RECEIVER==bob
 <- .send(RECEIVER,achieve,react_happiness_expression).
```

Figure 11: *The start of the beat Receiving the player*

As can be seen in figures 12, 13 and 14, the implementation of the characters' behaviors appear at their asl files. What is coded in the *beat_sequencer_agent* asl file is just used to trigger the behaviors of the characters. Consequently, in the *Mary and Bob* story implementation the characters are not autonomous.

```
// Receive the player if he has knocked on the door.
@behavior_receive_the_player
+!receive_the_player(HAS_KNOCKED)
 :HAS_KNOCKED == true
 <- !test_player_has_entered.

+!test_player_has_entered                                          3
 : player_has_entered(true)
 // The player has entered into the apartment
 <- dDialog.speak("Look at you! Come in. Bob, our guest is here.");
   .send(beat_sequencer_agent,tell,has_received_player(mary)).
-------------------------------------------------------------------
// The player has reacted with a happiness expression to
// the question "How are you?"
@behavior_react_happiness_expression
+!react_happiness_expression[source(S)]                            10
 :true
 <- dDialog.emote("happy","That's very good. ");
      .send(S,tell,reception_ended).
```

Figure 12: *Examples of Mary's behaviors for the beat Receiving the player*

```
// Bob receives the player.
@behavior_receive_the_player
+!receive_the_player                                               6
 :true
 <- dDialog.speak("Hello my good friend. It's very good to see you!
                   How are you?").
-------------------------------------------------------------------
// The player has reacted with a happiness expression
// to the question "How are you?"
@behavior_react_happiness_expression
+!react_happiness_expression[source(S)]                           10
 :true
 <- dDialog.emote("HAPPY","That's wonderful. ");
      .send(S,tell,reception_ended).
```

Figure 13: *Examples of Bob's behaviors for the beat Receiving the player*

```
@behavior_emoted_happiness_expression
+!happiness_expression(RECEIVER,emote(NAME))[source(S)]
 : true
 <- dDialog.emote(NAME);
 .send(S,tell,behavior(happiness_expression_realized(RECEIVER))).
-------------------------------------------------------------------
//For example, if the player say "I am fine" and emoted "SAD".
@behavior_emoted_and_spoken_happiness_expression_with_contradiction
+!happiness_expression(RECEIVER,emote(TEXT,EMOTE_DISCOURSE_ACT,
                                      NAME))[source(S)]
 : EMOTE_DISCOURSE_ACT == express_sad | NAME==bad
 <- dDialog.emote(NAME,TEXT);
 .send(S,tell,behavior(expression_with_contradiction_realized(RECEIVER))).
-------------------------------------------------------------------
@behavior_emoted_and_spoken_happiness_expression
+!happiness_expression(RECEIVER,emote(TEXT,DISCOURSE_ACT,
                                      NAME))[source(S)]
 : true
 <- dDialog.emote(NAME,TEXT);
 .send(S,tell,behavior(happiness_expression_realized(RECEIVER))).
-------------------------------------------------------------------
@behavior_spoken_happiness_expression
+!happiness_expression(RECEIVER,TEXT)[source(S)]
 : true                                                            8
 <- dDialog.speak(TEXT);
 .send(S,tell,behavior(happiness_expression_realized(RECEIVER))).
```

Figure 14: *Examples of Player's behaviors for the beat Receiving the player*

For [Wooldridge 1999, p.29, emphasis in the original], an agent "... is a computer system that is *situated* in some *environment*, and that is capable of *autonomous action* on this environment in order to meet its design objectives". However, in [Mateas and Stern 2000] view, having the believable characters strongly autonomous in a story world is problematic. In a story world, characters exist not just to convey their personalities; in fact, they should have the ability to take actions that move the story forward. Thus, the decision about the next action that an agent will take should depend not only on its internal state and on the environment condition, but on the current story state as well [Mateas and Stern 2000]. Therefore, similarly to Façade, in this experiment the characters are not strongly autonomous but rather provide the ways to achieve low level tasks [Mateas and Stern 2000]. The drama manager is the one that defines what the characters should do. In fact, the drama manager receives messages and perceives percepts in the environment (e.g. caused by a knock on the door) in a format of a behavior belief or percept that represents the way that the player has behaved in a particular situation. Based on the interaction with the player it decides how the NPCs should react. Therefore, to simplify the process, all messages that the player addresses to the NPCs are being redirected to the drama manager itself. In this case, the drama manager has all the power to decide how the NPCs must react.

The story implementation supports four types of input. The first type are the actions realized in order to affect the environment or character(s). For example, if the player character knocks on the door, the generated percept that is added to the environment is **behavior(player,door,knock)**. The second type of input is the textual input. The figure 15 shows an example of generated behavior belief for a textual input. Looking carefully the figure 11 you can see the definition of a plan to handle the addition of this belief (i.e. **behavior(player,bob,discourseAct(express_happy,player),"I am great today")**) to the belief base of the drama manager (see figure 11, step 7).

BOB: "Hello my good friend. It's very good to see you! How are you?"
PLAYER: "I am great today"
BOB:  "That's wonderful. "

(a) Example of textual input

User input
I am great today

Input parsed tree

```
(ROOT [36.647]
  (S [36.503]
    (NP [4.008] (PRP [2.746] I))
    (VP [26.192] (VBP [4.936] am)
      (ADJP [7.291] (JJ [6.513] great))
      (NP-TMP [4.228] (NN [2.676] today)))))
```

Typed Dependencies
[nsubj(great-3, I-1), cop(great-3, am-2), tmod(great-3, today-4)]

Discourse Act
discourse act type...: express_happy
subject...: player
direct object...: not defined

Behavior
behavior(player,bob,discourseAct(express_happy,player),"I am great today")

(b) The generated behavior belief

Figure 15: *Behavior belief for a textual input*

For the player's input to be considered by the drama manager, this input must first be transformed conveniently. For example, in the case of a textual input, the system tries to map the given input to a *discourse act*. The table 2 shows the discourse acts utilized in the *Mary and Bob* story.

agree	disagree	express_happy
express_sad	praise_aspect	praise_comportment
criticize_aspect	criticize_comportment	offer
drink	flirt	don_giovanni[10]

Table 2: *The discourse acts used in the Mary and Bob story*

The mapping of English text to a discourse act is accomplished by applying some rules that consider some information generated by the Stanford Lexicalized Parser[11] after parsing the input sentence (e.g. grammatical information in general, typed dependencies, etc.), and a XML file that maps words and emoticons to discourse acts. The figure 16 presents a fragment of the XML file used to map words and emoticons to discourse acts. The figure 17 shows an example of rule used to map a given textual input to discourse act. Note how the Stanford typed dependency representation (a binary grammatical relation held between a governor and a dependent [de Marneffe and Manning 2008]) play an important role in the definition of the discourse act. Also, it is important to point out that

[10] A special discourse act type where the player enters some sentences contained in the opera Don Giovanni of Wolfgang Amadeus Mozart with libretto by Lorenzo Da Ponte (e.g. "Voi non siete fatta per esser paesana").

[11] http://nlp.stanford.edu/software/lex-parser.shtml

the definition of the discourse act that maps to the given input may also depend on factors such as the story context. For example, in the beat *Mary makes advances to the player* a praise to Mary means that the player is flirting whereas in other beats a praise has not such connotation.

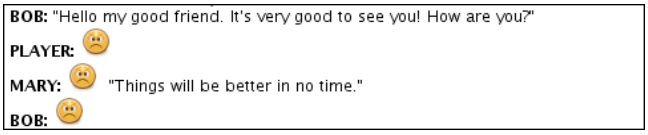
```
<discourse_act type="express_happy">
  <words>
    <word value="fine" />
    <word value="well" />
    <word value="great" />
    <word value="happy" />
  </words>
  <emotes>
    <emote value="happy"/>
    <emote value="laugh"/>
  </emotes>
</discourse_act>
```

Figure 16: *Example of discourse act in the Mary and Bob story*

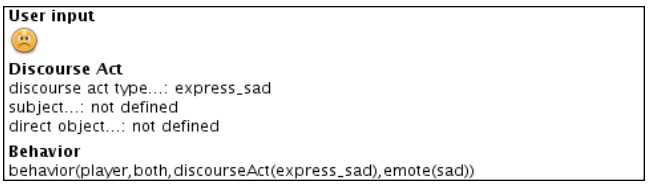
```
// Example: "I am great today"
// Typed dependencies:
// nsubj (nominal subject), cop (copula), tmod(temporal modifier)
// [nsubj(great-3, I-1), cop(great-3, am-2), tmod(great-3,today-4)]

// Generated by the parser from the given input
// nsubj(great-3, I-1)
nsubj_dep = "I"    // dependent
nsubj_gov = "great"    // governor
TYPED_DEPENDENCIES={nsubj, cop, tmod}
ADJECTIVES={word | word is tagged as JJ}

// Generated from the discourse acts xml file
EXPRESS_HAPPY={"fine", "great","happy", "well"}
EXPRESS_SAD={"sad", "bad","unhappy"}

// Verifying if the player is expressing happiness
IF   (nsubj_dep = "I")
   IF   (nsubj_gov ∈ ADJECTIVES
      AND cop ∈ TYPED_DEPENDENCIES)
         IF   ((nsubj_gov ∈ EXPRESS_HAPPY //e.g. "I am happy"
            AND neg ∉ TYPED_DEPENDENCIES)
          OR
          (nsubj_gov ∈ EXPRESS_SAD // e.g. "I am not sad"
            AND neg ∈ TYPED_DEPENDENCIES))
        RETURN discourse_act(express_happy, player)
```

Figure 17: *Mapping a given textual input to a discourse act.*

The third type of input is the emoticon. The figure 18 shows an example of generated behavior belief for an emoticon input. Notice that in such cases just the XML file is considered to define the discourse act for the input.



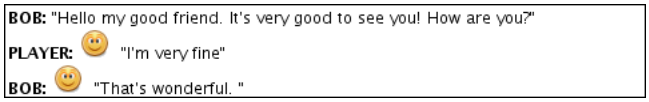
(a) Example of emoticon input



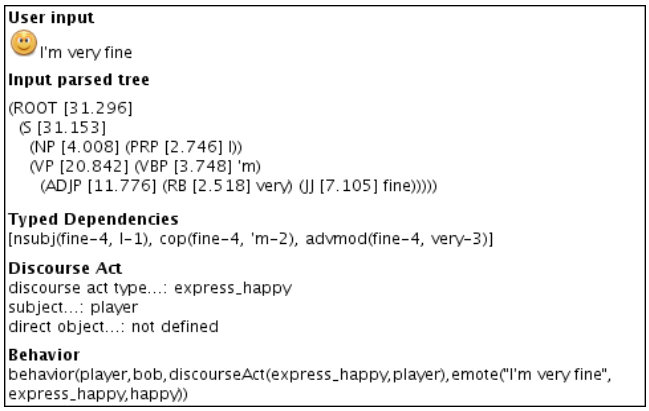
(b) The generated behavior belief

Figure 18: *Behavior belief for an emoticon input*

The last type of input is the textual input accompanied by an emoticon conveying a similar idea (or not). The figure 19 shows an example of generated behavior belief for a textual input accompanied by an emoticon. But what happens when the player's input consists of a textual input accompanied by an emoticon that contradicts the idea communicated by the textual input? The figure 20 shows an example where the player says "I'm very fine" at the same time he is crying. Note that the character Bob has reacted to this input in a different way (Bob appears confused). Contradictions are detected at plan level. Therefore, as can be seen in figure 14, behavior plans are added to the agents' asl files in order to treat them.

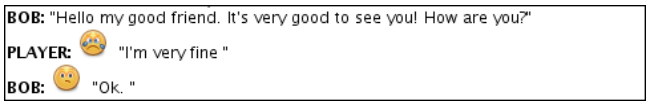


(a) Example of textual input accompanied by an emoticon

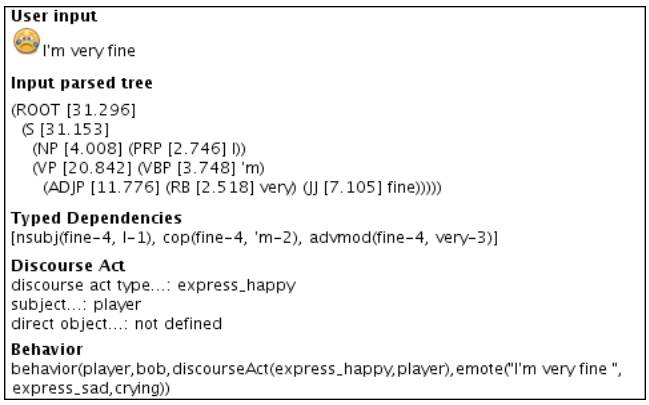


(b) The generated behavior belief

Figure 19: *Behavior belief for a textual input accompanied by an emoticon*



(a) Example of textual input accompanied by an emoticon that contradicts it



(b) The generated behavior belief

Figure 20: *Behavior belief for a textual input accompanied by an emoticon that contradicts it*

5 Conclusion

This work has presented *DGiovanni*, an open source multi-agent architecture for building interactive dramas. It has been developed in JAVA and uses the Jason's BDI engine, being available for download under the GNU General Public License Version 3. It borrows several ideas from the interactive drama Façade: the use of discourse acts, the use of the beat as the central building block of the interactive story, the use of the term behavior to mean a character activity, and the use of a drama manager to provide the high-level plot decisions. However, differently from Façade, for the developed architecture, none specific-purpose authoring language (such as the ABL and the beat sequencing language) has been created. Actually, the Jason's agent-oriented programming language has been utilized for performing the drama management and for authoring the characters' behaviors. In this regard, the architecture has adopted a multi-agent approach due to motives such as the resultant reduction

of coupling; the ability of agents to work in unpredictable, unreliable, and dynamic environments; and the more human-like manner that agents deal with problems which results from their proactiveness and reactiveness. In addition, Jason has been used because of several reasons: It is implemented in JAVA, thus multi-platform; it is available open source; it is highly customizable; it is a reactive planning language; the elegance of the AgentSpeak notation is particularly appealing to the symbolic representation of beats and behaviors; and lastly, because the use of the BDI paradigm may be interesting to the creation of characters that must exhibit a wide range of complex behaviors and interact with players that are usually unpredictable. The architecture also uses ontologies to support the creation of different stories and to feed the system with story-related information. Last, an experiment in building an interactive drama using the developed architecture has been presented as well.

In this context, the main contribution of this work is to provide an open source architecture that may be used in researches related to interactive drama. Additionally, it also shows how Jason can be used for performing the drama management and for representing beats and behaviors. However, there is still a lot of work to be done. One is to develop an interactive drama with better graphics and more interactive capabilities in order to evaluate the effectiveness and the scalability of the architecture. In addition, it would be interesting to have a program for helping in the authoring work, since the creation of beats and behaviors is very demanding. Also, the use of ontologies in the architecture must still be better studied. For example, the ontologies could keep much more information about the story such as those related to the characters' relationships (e.g. isFriendOf, loves, etc.). Moreover, in contrast to the realized experiment where the relation between the drama manager and the other agents is hierarchical (i.e. the drama manager sends commands to the agents), other schemes based on cooperation may be considered, such as the cooperative and the dialogical view of drama management [Szilas 2005]. Indeed, the hierarchical scheme has been adopted in the *Mary and Bob* implementation mainly to simplify the drama management. Therefore, since the architecture itself does not intend to impose the use of a hierarchical scheme, as a future work it is desirable to create a story where the NPCs have more autonomy. Finally, the Jason's customization capabilities can still be more exploited. For instance, the function that selects an intention for further execution can be customized in order to prioritize some of the intentions that are competing for attention.

References

BORDINI, R. H., HÜBNER, J. F., AND WOOLDRIDGE, M. 2007. *Programming multi-agent systems in AgentSpeak using Jason*. Wiley, Chichester.

BORDINI, R. H., 2010. Agentspeak [personal e-mail]. E-mail received on 4th November 2010.

DE MARNEFFE, M.-C., AND MANNING, C. D., 2008. Stanford typed dependencies manual. Available from: `http://nlp.stanford.edu/software/dependencies_manual.pdf`. [Accessed: July 2011].

FORSTER, E. 1956. *Aspects of the novel*. Mariner Books.

HAMMOND, S., PAIN, H., AND J.SMITH, T. 2007. Player agency in interactive narrative: Audience, actor & author. In *AISB'07 Symposium: AI and Narrative Games for Education*, 2007, Newcastle upon Tyne, UK. Proceedings of AISB'07, 386–393.

JENA, 2011. Common ontology application problems [online]. Available from: `http://jena.sourceforge.net/ontology/common-problems.html#aBox-tBox`. [Accessed: 22th July 2011].

LOYALL, A. B., AND BATES, J. 1991. Hap: A reactive, adaptive architecture for agents. Technical Report CMU-CS-91-147, Carnegie Mellon University, Pittsburgh, PA.

MAGERKO, B. 2002. A proposal for an interactive drama architecture. In *Proceedings of the AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment*, AAAI Press.

MATEAS, M., AND STERN, A. 2000. Towards integrating plot and character for interactive drama. In *AAAI Fall Symposium on Socially Intelligent Agents: The Human in the Loop*, AAAI Press.

MATEAS, M., AND STERN, A., 2002. A behavior language for story-based believable agents. Available from `http://users.soe.ucsc.edu/~michaelm/publications/mateas-aaai-symp-aiide-2002.pdf`. [Accessed: July 2011].

MATEAS, M., AND STERN, A., 2003. Façade: An experiment in building a fully-realized interactive drama. Available from: `http://www.interactivestory.net/papers/MateasSternGDC03.pdf`. [Accessed: July 2011].

MATEAS, M., AND STERN, A., 2005. Structuring content in the Façade interactive drama architecture. Available from: `http://www.interactivestory.net/papers/MateasSternAIIDE05.pdf`. [Accessed: July 2011].

MATEAS, M. 2002. *Interactive drama, art and artificial intelligence*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA.

MCKEE, R. 1997. *Story: Substance, structure, style, and the principles of screenwriting.* HarperCollins, New York.

MEADOWS, M. S. 2002. *Pause & Effect: The art of interactive narrative*. New Riders, Indianapolis.

NORLING, E., AND SONENBERG, L. 2004. Creating Interactive Characters with BDI Agents. In *Proceedings of the Australian Workshop on Interactive Entertainment*.

NOY, N. F., AND MCGUINNESS, D. L. Ontology development 101: A guide to creating your first ontology. Available from: `http://ksl.stanford.edu/people/dlm/papers/ontology-tutorial-noy-mcguinness.pdf`. [Accessed: July 2011].

PADGHAM, L., AND WINIKOFF, M. 2004. *Developing intelligent agent systems: a practical guide*. Wiley, Chichester.

PRESSMAN, R. S. 2001. *Software engineering: a practitioner's approach*, 5th ed. McGrawHill, Boston.

PRITNER, C., AND WALTERS, S. E. 2005. *Introduction to play analysis*. McGraw-Hill, New York.

RUSH, D. 2005. *A student guide to play analysis*. Southern Illinois University Press, Carbondale.

RUSSELL, S. J., AND NORVIG, P. 2009. *Artificial intelligence: A modern approach*, 3rd ed. Prentice Hall, Upper Saddle River.

RYAN, M.-L. 2008. Interactive narrative, plot types, and interpersonal relations. In *ICIDS '08: Proceedings of the 1st Joint International Conference on Interactive Digital Storytelling*, Springer-Verlag, Berlin, Heidelberg, 6–13.

SZILAS, N., WANG, J., AND AXELRAD, M. 2008. Towards minimalism and expressiveness in interactive drama. In *Proceedings of the 3rd international conference on Digital Interactive Media in Entertainment and Arts*, ACM, New York, 385–392.

SZILAS, N. 2005. The future of interactive drama. In *Proceedings of the second Australasian conference on Interactive entertainment*, Creativity & Cognition Studios Press, Sydney, 193–199.

TOBIAS, R. B. 1993. *20 master plots (and how to build then).* Writer's Digest Books, Cincinnati, Ohio.

WOOLDRIDGE, M. 1999. *Multiagent Systems: A modern approach to distributed artificial intelligence*. Mit Press, London, ch. Intelligent agents, 27–79.

ZANG, P., MEHTA, M., MATEAS, M., AND RAM, A. 2007. Towards runtime behavior adaptation for embodied characters. In *Proceedings of the 20th international joint conference on Artifical intelligence*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1557–1562.