Charles University in Prague

Faculty of Mathematics and Physics

DIPLOMA THESIS

Bc. Jakub Gemrot

Koordinace chování virtuálních lidí
Behavior Coordination of Virtual Characters

Department of software and computer science education

Supervisor: Mgr. Cyril Brom, PhD.

Faculty of Mathematics and Physic

Charles University in Prague

Study program: Computer Science, Theoretical Computer Science

2009

To my dearest … one day a boy and a girl will fall in love.

To my really patient parents … I'm sorry it took so long, I really am.

To Pogamut team members … working with you guys is fun!

To Cyril … ideas may be veiled in the fog, but you have a battery light.

… last but not least – thanks go to my alma mater – Charles University!

I declare that I have written this thesis by myself and that I have used only the cited resources. I agree with making this thesis public.

In Prague, 16. 4. 2009 Jakub Gemrot

Table of Contents

1 Introduction....6

2 Area of the thesis....8
- 2.1 Pogamut platform....10
- 2.2 Thesis's terms and abbreviations....11

3 Structure of the thesis....12

4 Goals 13

5 Virtual interactive storytelling....14
- 5.1 Believability of the story....15
- 5.2 Role-plot duality....15
- 5.3 Role definition....16
- 5.4 Story definition approaches....16
- 5.5 Chosen approach....18

6 Story authoring....21
- 6.1 Actors' behaviors....21
- 6.2 Abstraction of the story world....23
- 6.3 Story entities, facts and relations....23
- 6.4 Story actions....24
- 6.5 Story situations and boundary problem....24
- 6.6 Story authoring....25
- 6.7 Chapter conclusion....26

7 Architecture....27
- 7.1 Step 1 – Virtual environment....27
- 7.2 Step 2 – Sensing and acting in the environment....27
- 7.3 Step 3 – Actor's perception and story relations....29
- 7.4 Step 4 – Role definition....31
- 7.5 Step 5 – Plot definition....32
- 7.6 Step 6 – Story execution....32
- 7.7 Summarization....32

8 StorySpeak origin....33
- 8.1 BDI architecture....33
- 8.2 AgentSpeak(L)....34
- 8.3 AgentSpeak(L) extensions....37

9 StorySpeak....39
- 9.1 Additional StorySpeak extensions....39
- 9.2 StorySpeak plans and basic expressions....40
- 9.3 Additional expressions....42
- 9.4 Variables and unification....43
- 9.5 Calling Java methods, accessing Java fields....44
- 9.6 Template plans, plan delegation and parallel expressions....45
- 9.7 Failing plans....46
- 9.8 Subplans and handling of events inside instantiated plan....47
- 9.9 Plan annotations....47
- 9.10 Definitions....48
- 9.11 Interpretation algorithm....50
- 9.12 Extending StorySpeak....54

10 Storytelling framework.....55
10.1 Additional infrastructure work.....55

11 Evaluation.....57
11.1 Shared parts.....57
11.2 Story script 1 – Simple greetings.....59
11.2 Story script 2 – Greetings with replies.....61
11.3 Story script 3 – Story manager orders actors to party!.....64
11.4 Story script 4 – Ignorant Gregory.....66

12 Conclusion.....70

13 Future work.....71

Literature.....72

Appendix A – Related work.....74
Machinima.....74
Inscape.....74

Appendix B – The StorySpeak language grammar.....75

Appendix C – Enclosed CD.....80

Název práce: Koordinace chování virtuálních lidí
Autor: Jakub Gemrot
Katedra: Katedra software a výuky informatiky
Vedoucí diplomové práce: Mgr. Cyril Brom, PhD.
e-mail vedoucího: brom@ksvi.mff.cuni.cz

Abstrakt: Tato práce je o specifickém přístupu ke koordinaci chování virtuálních agentů. Každý virtuální agent je schopen jednat sám za sebe, ale také může být kdykoli veden některým z koordinačních agentů. Tento přístup je navrhnut speciálně pro oblast interaktivního storytellingu, kde jednotliví agenti - herci jsou vnímáni pouze jako loutky, které jsou ovládány abstraktním agentem - režisérem. Kontrolní mechanismus agentů je založen na BDI architektuře. Zejména na jedné z její implementací a to jazyku AgentSpeak(L). Jazyk AgentSpeak(L) je rozšířen o šablonované plány a nový mechanismus exekuce plánů, který umožňuje zmíněnou kontrolu agentů - herců.

Klíčová slova: virtuální interaktivní storytelling, BDI, rozšíření AgentSpeak(L), definice příběhu, provádění příběhu

Title: Behavior Coordination of Virtual Characters
Author: Jakub Gemrot
Department: Department of Software and Computer Science Education
Supervisor: Mgr. Cyril Brom, PhD.
Supervisor's e–mail address: brom@ksvi.mff.cuni.cz

Abstract: This thesis is about specific approach to the behavior coordination of multiple embodied virtual agents. Agents may act for themselves or be controlled directly by bodiless coordination agents. This kind of approach is designed for the area of interactive storytelling, where the actor agents are viewed as a string puppets that are controlled by the abstract director. The control mechanism is based upon the BDI architecture and the AgentSpeak(L) language that is extended with template plans and new plan execution mechanism that allows the directing of other actor agents.

Keywords: virtual interactive storytelling, BDI, AgentSpeak(L) extension, story definition and execution

1 Introduction

Computers are truly amazing. They seem to have many applications as they are part of our daily life. They plan air traffic, observe nuclear reactions, keep our keys to bank accounts and control dish-washer machines at night, so we may go to bed earlier. No one is surprised that they are used for practical reasons, but computers are used also for their artistic potential. However the idea may seem to be bizarre, computer programs are capable of generating beautiful things such as songs and pictures. I am not talking about 3D modelers or audio composers as they are only software tools and not generative systems. There are examples of works, David Cope's experiments in musical experiments[1] or evolutionary art (see fig. 1), where computers are used in new inspiring ways.

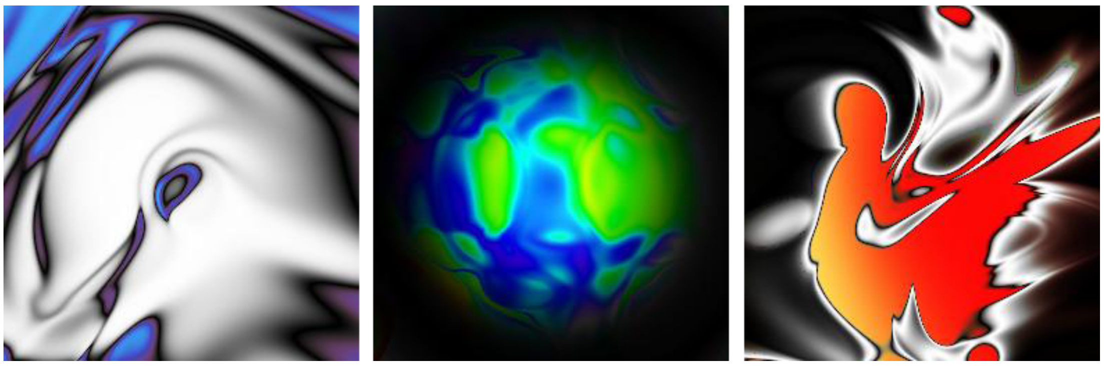


Figure 1 – Pictures from picbreeder.org[2] – Dolphin, Habitable planet and Rock drummer – that were evolved by neural network with human teacher.

It is no surprise that computers made also their way to the field of storytelling. Not only that they allow to play movies, but they are bringing interaction. Imagine that you may influence the course of the story of Hansel and Gretel. Have you ever tried to think about: What would happen if the wicked witch escapes the oven (she is a witch after all) and starts to chase Hansel and Gretel? Would the poor children hide themselves in the nearby bush? Or would they just run as fast as they can – but what if the witch have a broom? Will the kids manage to escape then? Or the story turns into a horror, which – let's admit it – fairy tales are usually not far from.

This interactivity we brought to the story brings up the question – who knows such a version of Hansel and Gretel story, where the wicked witch escapes the oven and even has the Nimbus 2008? If a storyteller was a man, it would be left to his or her imagination as it is the case of role–playing games like Dungeons&Dragons. D&D is a board game where players are involved in fictional story. Each player of D&D speaks for his own imaginary character except for player that is the game master. The game master has the role of the storyteller that tells the story to other players, decides effects of players' actions and speaks for all other non-player characters. His role is to maintain the believability of this evolving and ever–changing story. That is for humans, but how can one describe possibilities in the

[1] http://arts.ucsc.edu/faculty/cope/experiments.htm [15. 4. 2009]
[2] http://picbreeder.org/imagedisplay.php?type=RANK [15. 4 .2009]

story so the computer could unfold them? How to describe the possibilities in the story line? This thesis is suggesting a specific approach to the story definition allowing the user to define plans for autonomous actors as well as for bodiless director agent. The director agent may interrupt actors' plans at any given time allowing the author of the story to coordinate them and express plot events.

2 Area of the thesis

Virtual storytelling concerns itself with unfolding of a story inside a specific virtual environment be it textual, 2D or 3D world. The story is typically told by a number of computer–controlled actors, which inhabit the world. The field is interdisciplinary, bringing together researchers from movies [Clarke01], psychologists [Ruth06], computer scientists [Bae08] and linguists [Kopp05]. Its roots lie within automatic generation of story scripts using planners [Turner92]. As time went by, and computers became capable of 3D visualization, this idea evolved into orchestration of computer–controlled virtual actors. Virtual actors, faithfully visualized, may tease the viewers interest and fill them with anticipation of deep stories as viewers are used to see in cinemas and perhaps more. Viewers are now allowed to interact with actors, ceasing to be only passive consumers. Viewers-players are given a freedom to roam inside the virtual environment in search for their own and unique story experience, which is determined by the sequence of actions they take inside the world (or their absence). It is exactly this interactivity that brings new opportunity for authors to produce emotionally rich stories as well as burdening them with a new problem. Books are read from the beginning to the end as well as movies are watched, so the authors have complete control over the story line. Interactivity prevents this. The interactive story is not read or watched in author's intentional sequence but it is interleaved with players' actions and decisions that can prevent the unfolding of storyline as intended by the author. Perhaps the author should the abandon his strong intentions in the first place.

Truly the best analogy for the virtual interactive storytelling would be computer games, specifically a role–playing games (*RPGs*, see fig. 2) where players are in control of an avatar that is thrown into the middle of a specific story.

Figure 2 – Neverwinter Nights 2, typical example of RPG game, where a player controls the party and performs quests that allow him to advance in the story.

Players find themselves in a living world with tens of characters, who guide them through the intended story line by giving players quests to complete, which direct their attention. Still, the most of RPGs are considered to be boring because the set of players' possible actions is limited which makes the story to be almost linear. This approach is used to avoid the combinatorial explosion of possible situations that may arise in the world allowing the game developers to predict situations the players may experience. But there are also examples of games that leave players in the world on their own, allowing them to literally live inside it [Fable 2, Fallout 3, World of Warcraft], explore the world, own houses, even have wives and children. And it is this freedom the virtual storytelling would like to give its players. The possibility to be a part of the story, interact with other actors while being guided on the story line forward, which makes the fictional Star Trek holodeck[3] idea the holy grail of the field.

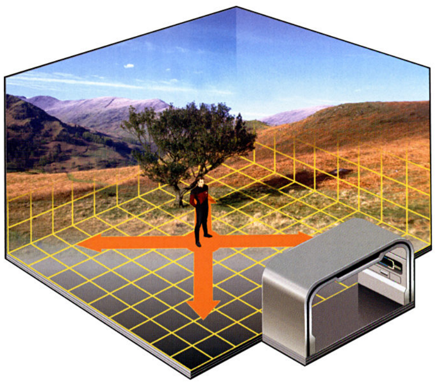


Figure 3 – The Star Trek holodeck provides 3D holographic visualization of the virtual world that allows people to walk inside the environment.

But why is the idea of Star Trek holodeck (see fig. 3) so appealing? What would people seek by entering it? Entertainment, chance to be somewhere else, become someone different – even if just for the moment. People are aware that there are so many places in the world, so many things to try that they can not see and experience them all. Human life is just too short for that. So we are eagerly reading detective stories, romantic novels, watching horror movies or historical dramas searching for new sensations while always asking ourselves a question – how would it feel to be inside such story? And it is exactly the experience that the holodeck in Star Trek is offering, an opportunity to live through the story on your own and to make decisions on your own, be Robin Hood who fights the Sheriff of Nottingham, be Sherlock Holmes and uncover the mystery of the Hound of the Baskervilles or just live one day together with Friends.

Will the entertainment industry be the only one who would benefit from this device – the holodeck? Certainly no. Policemen, firefighters, rescue workers they all could undergo many training scenarios in virtual environments. They could face robbers, learn the drill during extinguishing forest fires or try to save as many lives as possible during the natural disaster. The possibilities would be numerous, what U.S. Army at least is aware of as they are financing developments of virtual simulators for combat training, learning languages or developing the social skills that are needed during the missions in countries with foreign culture [Johnson07].

[3] http://en.wikipedia.org/wiki/Holodeck [16. 4. 2009]

Figure 4 – Header of the Tactical Language web page [4] where they explain how the training simulations in virtual environments may help with understanding of foreign languages and cultural nuances.

Strangely, there is not many differences between training scenarios and fairy tales. Of course that they are totally different – fairy tales are told to children to appease their fear from fantasies, that are inhabited by bogeymen and training simulations must challenge the trainee with problems she is likely to face during her work[5]. But looking through the glasses of virtual storytelling, we find out that similar mechanisms are working in both of them. One have to present the audience an environment (be it an Iraqi town or the gingerbread house) and populate the environment with animate objects, which tell the story. But how the animate objects will be controlled?

Let's pretend for a moment that we have a holodeck that is capable of displaying 3D objects and animate it to any extents. This gives us the way how to take the audience into the environment of the story. To complete the interactive part of the storytelling we need a way how to say "computer, the wicked witch is living inside this gingerbread house and she will try to catch and eat anybody who lays a finger on gingerbread from her house's walls" defining the role of the witch in the story. Then we will need to specify the challenge for the player by saying "computer, tell the players that their aim is to stole as many gingerbread as possible before the wicked witch catches one of them". And we hope the computer to reply with "command accepted, computing the wicked witch behavior … finished, would you like to validate the story by playing it?"

Certainly, automatic natural language understanding is not on the level for this kind of interaction with the computer to be possible, but do we have any other way how to tell that to the computer?[6]

2.1 Pogamut platform

Virtual interactive storytelling application can not be created without virtual environment. Implementation of the 3D engine is a huge task. Fortunately, the idea to use an existing 3D virtual world for embodied virtual agents (actors in our case) is not new [Adobbati01]. There already exist projects that are offering various APIs[7] for controlling virtual characters *(avatars)* inside a specific virtual environment. One of them is the Pogamut project that is being developed at our faculty in Prague.

[4] http://www.tacticallanguage.com/ [16. 4. 2009]

[5] For instance, the illustration of an environment and a story will be totally different with gingerbread and the witch who imprison the players on the one hand and Shia doctor who refuses to abandon his clinic even though there will be a bombing raid on the other hand.

[6] On the higher level of abstraction then the C or Java language.

[7] Application programming interfaces.

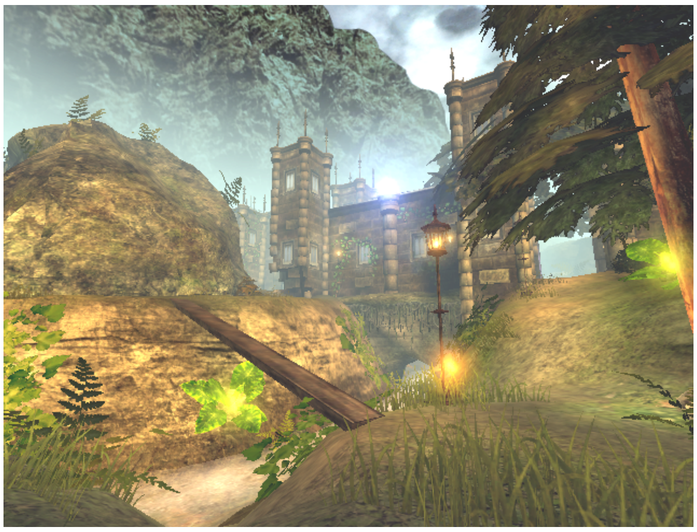


Figure 5 – Virtual environment of the UT04

Pogamut is a platform that allows controlling avatars inside an environment of the commercial game Unreal Tournament 2004 (*UT04,* see fig. 5) using Java. Pogamut is designated for research projects concerning investigation of behavior of human–like virtual agents and the education of undergraduate students. UT04 offers adjustable human–like virtual 3D world together with a lot of different locations, library of predefined items and a map editor.

The initial aim of the platform was to provide rich environment of UT04 for academic community. The platform was already successfully used in a few master thesis by [Gazolla06] [Kadlec08] and it has been used in international competition BotPrize at Australia [BotPrize08].

The platform has also a potential to be suitable for experiments in virtual interactive storytelling. The map editor may be used for the authoring of the virtual environment. The UT04 rendering engine is customizable and allows using custom 3D models and animations for actors. And the Pogamut library may be used to control these actors.

However, the platform is currently lacking a framework that would be built over the Pogamut providing a way for story authoring.

2.2 Thesis's terms and abbreviations

I would like to note at this point, that the whole thesis's concern is “virtual” storytelling. If I further speak about environments or actors, I will always mean it as virtual environments and virtual actors, etc.

Throughout the text I will often use the word *author* to refer to the creator of the story. Usage of the word *author* should not invoke the feeling that an author is not a programmer because the presented framework is suitable only to authors who have some background in computer science as well.

The thesis's concern are *virtual interactive stories*, I will abbreviate them as *VIS*.

I will use the term *virtual environment (VE)* for some specific implementation of the 3D rendering engine (e. g. Unreal Tournament 2004). The graphical appearance of actors inside a chosen VE are called *avatars*.

I will also use terms of actors' roles and actors' behavior interchangeably as the role of the virtual actor is expressed by the actor behavior during the execution of the story.

3 Structure of the thesis

The structure of the thesis is as follows.

Chapter 4 is presenting the goal of the thesis and divide it into several subgoals.

Chapter 5 is presenting the virtual interactive stories and their main problem of narrative-interactive tension.

Chapter 6 is discussing authoring of actors' roles and the definition of the story world. The story authoring is divided into six steps that are further discussed in chapter 7.

Chapter 8 and 9 concern themselves with the main part of the work and that is StorySpeak language that is used to describe situation/behavior pair during the role definition.

Chapter 10 summarizes how the storytelling platform supports the author.

Chapter 11 evaluates the platform with a few story scripts showing its features.

Last two chapters conclude the thesis.

4 Goals

The goal of the thesis is to provide a framework for behavior coordination of virtual actors for the Pogamut platform. This framework makes the first step towards the framework for the definition of interactive stories providing a means to perform short scenes with several virtual actors. To fulfill this goal we need to:

1. introduce virtual interactive stories and present a story definition and execution problem,
2. discuss existing approaches to the story definition (and execution),
3. describe a chosen approach,
4. identify steps of the story authoring in the context of chosen approach,
5. create a framework for behavior coordination of virtual actors,
6. evaluate the framework with a few story scripts that will present its features.

The thesis is focusing on the definition of actors' behaviors as sequences of actions and does not concern with graphics, gestures, mimics, dialog management or natural language processing.

5 Virtual interactive storytelling

Virtual interactive story (VIS) is a non-linear story that is told by virtual actors in virtual 3D environment to a player that is present inside the environment either as an observer (ghost) or as a player-actor. To make the story interactive, the player must be able to interact with the environment. The interactivity is bringing a problem for the story. The players' actions can not be foreseen therefore they may interrupt the story execution at any given time. Depending on the freedom of the player, it may have various effects on the course of story. The player may be given one of the three degrees of freedom:

1. The player can be only a passive observer that is allowed to watch the story from different places or from the eyes of different actor. E. g., the player may decide to watch the story from the Hansel's eyes – see what the actor may see, or just fly with the camera over the gingerbread house. Although the story can not be called interactive in this case, it brings one question: Will the player see all important events in the story this way? After all, the player might direct the camera at the first tree and might ignore the story completely.
2. The player may be an active observer that may influence actors or the environment. He could be given a way to change the mood of actors, alter their goals or give them additional items. How the story of Hansel and Gretel would continue if Hansel was given a pair of lock picks to get out of the cage at midnight?
3. The player may be present in the story as another actor. The player can play the role of Hansel or the wicked witch. How should Hansel and Gretel behave when they found the empty gingerbread house because the player in the role of wicked witch had decided to find some herbs for a love potion?

If the story is constructed as a series of events, that is a series of actors' actions, then it will result in an unbelievable story. The passive observer may miss the part where the wicked witch imprisoned Hansel and Gretel because he admires meadow full of flowers. The active observer may be surprised that Hansel, who was given lock picks, is not using them. And in the last case, the story could not even be planned this way as we can not force the actor to behave according to the script. The linearity of the story has to be abandoned.

Nevertheless, the author's intentions are not only to present a believable story world, but also to tell the story. The author intends the player to experience different situations. The fear from the wicked witch when the player as Hansel is running from her. Bravery of Robin Hood who was given a sword by the player allowing Robin Hood to escape from sheriff's guards. Or just observe Ross who is trying to be alone with Rachel. These author's narrative intentions face unanticipated actions of the player. The author wants the player to experience some events in the story, but the player has a freedom to prevent these events from happening or triggers them in different order. This is called narrative-interactive tension and this is the main problem of the virtual interactive storytelling. The role of virtual storytelling framework is thought to be a mediator of this tension [Magerko05].

How to mediate the narrative-interactive tension? Before I describe three approaches that are being widely used, I would like to present a bit weird analogy between an interactive story and a computer program. The computer program can be seen as an interactive story. Threads are actors, data objects items and the threads' code that transforms data objects are actors' roles. We may start the program in debugging mode and watch the story of the user input that is being transformed to the final output. How many if-then-else statements the data will face before they are transformed? Each such if-then-else condition unfolds a different story and the program produces different output. If the program receives unanticipated input, it will crash with segmentation fault or produce some strange output.

Although the analogy is far-fetched, we should have two things in mind: 1) if the player perform an unanticipated behavior (there is no behavior written for it), the story will always fail, 2) the more if-then-else conditions the interactive story contains, the more stories as sequence of events it contains.

The former is suggesting that, if we give too much freedom to a player, there always will be cases we have not thought of and the feeling from the story will disappear in the moment the player does not fall behave as we anticipated. On the other hand, limiting players action weakens the interactivity.

The latter tells us that, what we do not encode to the program of the story, it will not simply be there. The more non-linear we want the story to be, the more code we have to write.

This is a second problem of the virtual interactive storytelling. How the story should be encoded to express the most story lines with the fewest lines of definition as possible?

5.1 Believability of the story

Apart from the narrative-interactive tension and the non-linear story definitions problems, there is another point that must be addressed by the virtual storytelling framework. That is the believability of the story and its actors. If the player see Hansel to be a crybaby in the dark forest, it will be strange of Hansel to try to escape from the witch at all cost. The story will fail to be believable and it will harm the player's experience.

5.2 Role-plot duality

There is a tight correspondence between a definition of a plot and definition of roles because the story is always told through actors' actions *(role)*. We may sketch the event as “Hansel is imprisoned by the witch.” but we will have to specify what the actor should do during the event after all. We may say that the plot implies roles. On the other hand, when we see the witch dragging poor Hansel into the cage, locking him up while yelling on Gretel to clean the house, we quickly recognize the event in the story. Therefore roles imply the plot as well. As the computer who executes the story can not come up with actors' actions it does not know, there are always roles that are being defined. This means that authors are defining short sequences of actors' actions that are being triggered during the execution of the story.

5.3 Role definition

What does the term role mean? I see it as an analogy of the social role. Sociology defines the term social role as a set of 1) obligations, 2) connected behaviors and 3) rights as conceptualized by social actors in social situations [Keller06]. We may analogically define the role of the wicked witch following the definition from sociology: 1) she is bound to protect her house 2) by catching and imprisoning everybody who is trying to steal gingerbread from her house 3) using spells and traps. The last part of the sociology definition tells us that the actor has to behave according to the context of some situations. This means that the wicked witch should exhibit different behavior when she is catching Hansel and Gretel, then catching the highway robbers. Hansel and Gretel are children and there is no need to use magic, but dealing with armed robbers requires different approach. Therefore to define the actor's role means to identify the set of story situations an actor may face during the story and specify a behavior for each of these situations. The more story situations the actor's role has behaviors for, the better acting the actor will exhibit. The actors' role will always be a list of pair situation,behavior.

5.4 Story definition approaches

There was a lot of discussion in the storytelling community how the virtual interactive story should be defined. The consensus has not been found yet, but two different approaches has been identified. There are two different kinds of approach to the story definition: 1) author-centric and 2) character-centric. The former is thought to create stories with a strong plot coherence but poor character believability and vice versa [Riedel03].

Author-centric approach

Author–centric approach is a story authoring concept that is trying to formalize the author mental process during the story construction. The story is being constructed by sketching the plot as a sequence of events on the storyline. The author is meant to specify important events that should happen inside the story and the *story manager* should cover the rest.

Having the storyline definition from the author, the story manager is meant to plan the actions of actors according to those events and control these actors during the story execution. Actors have no own reasoning mechanisms

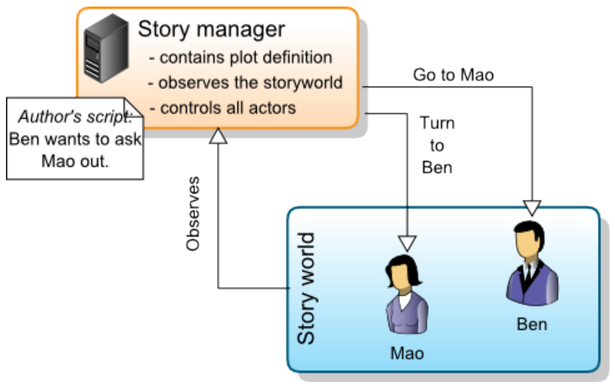


Figure 6 – The story manager is guiding Ben and Mao according to the author's script. The story manager has to supply every action for them.

The story is defined as an initial state of the story and a list of events. For instance, the story of Hansel and Gretel could be describe like this:

1. story starts how the father leaves Hansel and Gretel in a forest,
2. Hansel cries and Gretel is trying to calm him,
3. they go through the forest,
4. they find the gingerbread house, ...

Certainly, such definitions have to be formalized for the planner. The events would have a form of states of the story. Except of the list of events, the author would have to define a list of actions that actors may perform. The planner would have to plan all actions for actors to satisfy the ordering of events (story states).

This approach has a problem with narrative-interactive tension during a story execution – the story manager must be able to change actors' actions according to player's actions. These actions might be even severe, e. g., the player kills one of the main actors or sells an important item to an enemy. Nevertheless, if such real–time planner that copes with unanticipated situations is devised, it will be the best mediator of the narrative-interactive tension.

Character–centric approach
Character-centric approach to the story definition models each actor as an autonomous agent [Wooldridge95] with own goals. The story, as a sequence of events, should emerge from the interaction of actors. There is no central story manager that observes the scene and orders the actors what to do or synchronizes their behaviors. Example of this approach is [Cavazza01]. Cavazza models the actor's mind as an HTN planner making HTN plan a role of the actor. Such approach is successful until the actors need to do a joint–behavior. Introduction of joint-behaviors would need to create some synchronization mechanism that would ensure that all actors plan joint–actions at the same time. This leads to the idea of combined approach.

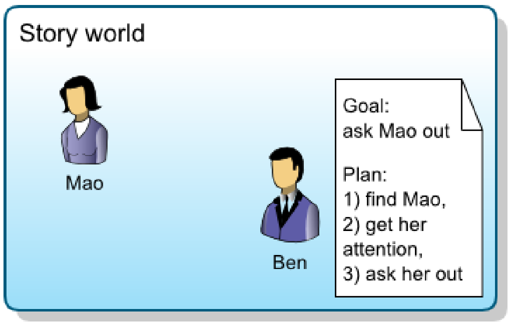


Figure 6 – Actors are modeled as an autonomous agents with own goals and plans.

Combined approach
Combined approach embraces both author–based and character–centric approach. It leaves autonomy to characters that may be modeled independently while sustaining a story manager that is omniscient. The story manager may be used to coordinate behaviors of actors by altering their goals to direct their efforts in order to maintain the pace and believability of the story. Example of this approach may be the work of

Magerko and his IDA (Interactive Drama Architecture) [Magerko05]. Actors in IDA are semi–autonomous. They may pursue their own goals but they have to obey the director agent that directs the story according to story content and story structure. Magerko model story content as a set of plot points which are defined as doublets `<precondition, action>`. Story structure then defines directed edges between plot points creating a partial ordering of plot points. Partial ordering of plot points is defining every possible topological ordering of plot points thus defining every possible story. Additionally his story manager maintains a player model and predicts the possible player behavior thus it can take an action whenever a player actions threatens the intended storyline. Unfortunately, Magerko's work was not made public and can not be evaluated.

5.5 Chosen approach

I believe that the storytelling framework needs to combine both approaches. There are two reasons:

1. Author-centric approach requires a real time planner that needs to be really fast to be able to react to players' actions. If the planner is not supplied, then the author has to write the actions nevertheless.
2. It is difficult to represent joint-behavior with character-centric approach.

The former can be solved by using reactive planners with predefined plans for the actor (character-centric approach). The latter can be solved by introducing the story manager that may control actors directly when coordination of two or more actors is required (author-centric approach).

Presented storytelling framework will embrace both approaches by allowing the author to provide plans for both actors and story manager. It will allow the author to write plans from the perspective of an actor (character-centric approach) and also from the perspective of the story manager (author-centric approach).

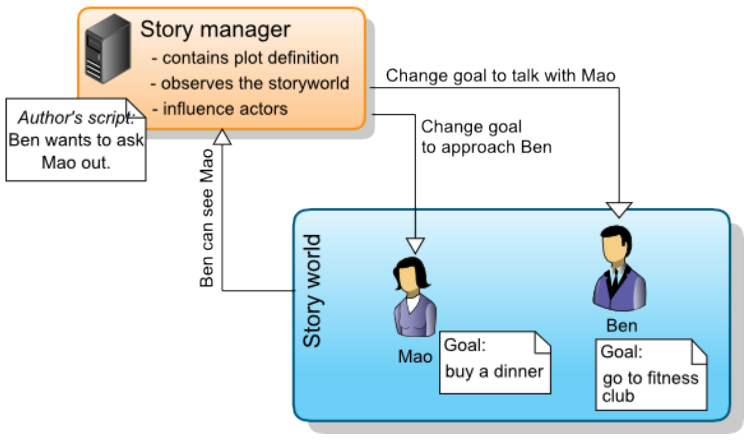

Story manager
- contains plot definition
- observes the storyworld
- influence actors
Author's script:
Ben wants to ask Mao out.
Change goal to talk with Mao
Change goal to approach Ben
Ben can see Mao
Story world
Mao
Goal: buy a dinner
Ben
Goal: go to fitness club

Figure 7 – Actors have own goals, plans and they act as autonomous agents. The story manager observes the story world and performs joint-behaviors with actors to unfold the story.

To support the definition of the believable characters, the actors will have their own perception of the story world. Plans for actors will be executed always in the context of facts the actors perceive.

The story manager will have an access to the complete state of the story in contrast with actors. This will allow the author to express joint-behaviors as story manager plans. Additionally, joint-behaviors should be also used to express events of the plot.

The analogy with the sociology role (ch. 5.3) tells us that all behaviors are expressed always in the context of some situations. The wicked witch will behave differently when facing robbers then when dealing with Hansel and Gretel.

I see the actor's role definition as a definition of role pair situation–behavior. The author would have to be able to define pair situation–behavior. Whenever some situation is recognized, the behavior of the actor should be switched.

The same mechanism could be used also for the definition of the story manager. The story manager has to act whenever some situation is encountered.

Story situations

The situation can be thought of as a state of the story that satisfies the situation's conditions. For instance, the situation can be “the wicked witch saw Hansel who was eating gingerbread” or “Hansel and Gretel were alone in a forest”. The story situation is based on *story states' values*. Additionally, the situation is usually triggered by some event inside the world. E. g., “the wicked witch saw”. The situation is also described by *events that have just happened*.

Examples of such values may be the wicked witch's position, description of the gingerbread house, what items Gretel has, what Hansel is doing, list of objects the wicked witch may see, etc. These values may be discrete (Gretel has a knife vs. Gretel does not have a knife) or continuous (position in the 3D world).

Events are recent changes of story states' values. Example is “the wicked witch saw Hansel”. This is an event that Hansel has been added to the wicked witch's list of visible objects. The framework will have to observe such changes and produce events as they appear.

The attention should be paid to the representation of the story world as this implies the way how situations' conditions will be written. For instance, if the objects that the wicked witch may see, is represented as a list, then the author would have to iterate over such a list every time he wants to check whether the wicked witch can see Hansel.

There is a question whether the situation's query should be evaluated over the complete state of the story world or only over its limited subset. 1) Should every actor be omniscient having always access to all story facts' values and relations that currently hold or 2) should every actor have an access only to a limited set they perceive?

The first option could be considered cheating as an actor will perform behavior that is based on facts it can not know. It may be a coincidence that the wicked witch appears behind my back for the first time I try to steal gingerbread, but after the third attempt I will start to be suspicious.

On the other hand, the behavior of the actor that is based only on its observation may prove to be limited. We will not be able to create surprises such as meeting of two actors on the corner of the street as both of them do not have information about the location of the other one. This supports the idea of the omniscient story manager that should guide actors in such situations and the combined approach to the story definition.

Appropriate behavior

Behavior is defined as actions or reactions that a virtual actor exhibits in relation to its virtual environment. What is an appropriate behavior depends on the author aim and his concept of the story. If the author models a role of the wicked witch, he will want her to be aggressive and blunt. If the author wants to surprise a player, he will make the witch kind in the beginning and aggressive later when the player will feel safe.

Implemented parts

Provided description of the chosen approach creates a frame for the implementation of the behavior coordination of actors. The aim of the thesis is not to provide the complete storytelling framework that supports the author along the way of the story construction but to provide a way for the behavior coordination. Nevertheless, the solution should be created with the whole picture in mind.

6 Story authoring

This chapter is discussing the authoring of actors' behaviors. Behaviors are categorized as interactive or sequential. The storytelling framework should allow actors to switch between behaviors of these two types. It is illustrated on the example of the story of Hansel and Gretel.

6.1 Actors' behaviors

The story authoring happens in two steps. Firstly, the author has to sketch the story into a sequence of story events. Secondly, the author rewrites the story using a definition language of a storytelling framework.

Let's say that the author is creating the story of Hansel and Gretel and intends the player to be Gretel. The author sketches the story to follow this sequence of events:

1. Hansel and Gretel are left in the woods,
2. they find the gingerbread house,
3. the witch imprisons Hansel and enslaves Gretel,
4. Gretel tricks the witch, sets free Hansel,
5. they are trying to escape the witch.

Now, the author has to categorize behaviors of virtual actors into two categories:

1. interactive behaviors
2. sequential behaviors

The former behavior is expressed when the actor needs to interact with the player as it must react to player's actions. The latter behavior is expressed when virtual actors are interacting with each other. Creating interactive behavior is much more complex then creating sequential behaviors. This should be clear as sequential behaviors are just the list of actions that actors should perform, while interactive behavior must contain many decisions points that reflect the possibilities of players' (or other actors) behavior.

For instance, the sequential behavior can be used when the witch wants to imprison Hansel as it is a behavior that is expressed only by two virtual actors. This is the case when the storytelling framework should allow the author to express this behavior as simply as possible.

On the other hand, the author will need to create an interactive behavior for the witch when she will need to interact with the player in the role of Gretel. Reactive plans are widely used for this approach. Reactive plans can be thought of as a list of if-then-rules, which are being periodically evaluated. If the storytelling framework is written only as character-centric, then it will force the author to express sequential behaviors inside these if-then-rules (see fig. 8).

Witch's plan	*Hansel's plan*
1. yell at Hansel	1. wait for witch's yelling
2. wait for Hansel to come near me	2. go to the witch
3. grab the Hansel	3. wait for the witch to grab me
4. go to the cage	4. go to the cage
5. order Hansel to go inside	5. wait for the command
6. wait for Hansel to go inside	6. go inside the cage
7. lock him up	7. cry

Figure 8 – Two plans that must be created in order to perform joint-behavior in character-centric approach using interactive behaviors.

Figure 9 shows how the same behavior may be expressed with author-centric approach and sequential behavior.

Joint-behavior of the witch and Hansel

```
1.     witch: yell at Hansel
2.     Hansel: go to the witch
3.     witch: grab the Hansel
4.     witch + Hansel: go to the cage
5.     witch: order Hansel to go inside
6.     Hansel: go inside the cage
7.     witch: lock him up, Hansel: cry
```

Figure 9 – Plan that coordinates behaviors of the witch and Hansel written as sequential behavior for both actors. It contains half lines then the same behavior written with character-centric approach.

It may seem that the author-centric approach should work the best for the storytelling applications as we may create one reactive plans for all actor. That is not true. If we would like to express all actors behaviors within one plan we would face the combinatorial explosion of situations that results from the combination of all actors' plans into one big plan. If we think about the character role as a list of if-then-rules (condition->action) and role *A* has *N* rules and role *B* has *M* rules. If we merge these two roles into one, we will need to provide *N*M* rules in the worst case. After checking the condition for the role A we will have to go through conditions of role B.

Therefore the combined approach should be used to spare the author of unnecessary work. The storytelling framework has to allow interleaving interactive behaviors of respective actors and sequential joint-behaviors.

Chapter 9 shows how the presented framework and its language StorySpeak supports interleaving of these two types of behaviors.

6.2 Abstraction of the story world

There is a gap between definition of actors' roles (behaviors) and the virtual environment. The author could use the interface of the virtual environment only to define actors' behaviors but this interface will be rather low-level. For instance, the Unreal Tournament 2004 provides only simple actions such as “move directly to”, “jump”, “say”. If the author wants Hansel to follow Gretel, he will need to decompose such action further to use only low-level actions that are provided by UT04. Such decomposition will face implementation details, e. g., obstacles avoidance. The objective of the author is to create the story and not to compute movement vectors for actors. Therefore we have to provide a layer of abstraction between virtual environment and the author. We should create an abstract world of the story *(the story world)*. This layer abstraction should not be only in the terms of actor's actions but also in the terms of the story world perception due to the same reason. For instance, 3D virtual environment represents the locations of actors in absolute terms as triples *x,y,z*. The author should not count distances in space between locations of two actors, when he wants to find out whether they are near to each other. The storytelling framework should supply him with these facts automatically.

Unfortunately, we can not really say which facts about the world the author will need during the definition of actors' behaviors. Therefore the storytelling framework will have to support their definition and it will become the part of the story authoring process – to formalize the author's story world into story facts that describes it.

6.3 Story entities, facts and relations

The story world will surely contain places, objects[8] and actors.

Every place, object or actor will have some kind of characteristics - the set of properties that defines the *story entity*. We will call them story facts. The actors' characteristics may be their positions, what items they currently have or their current mood. The objects' characteristics may be their weight or form. The places may be described by their boundaries within the virtual world and be labeled as forest or the gingerbread house. They are all up to the author to define and author should define all story facts that are relevant to the story. They will make the basis, which the author will define situations upon.

For instance, if the author will need to express the situation “when the Hansel is near the door to the gingerbread house” it will need to know: where is the gingerbread house, where it has door and what is the position of Hansel. Additionally, the relation “is near” would have to be expressed.

Apart from the base characteristics, there could be relations between entities. The wicked witch's characteristics may be “at position x,y,z of the world” and also “at the gingerbread house”. But the gingerbread house is one of the places in the world thus it will have its own characteristic – “is the cube of virtual space of coordinates x,y,z,x,y,z”. This makes “at the gingerbread house” a relation between the wicked witch and the gingerbread house because it may be inferred from the facts that the position of the witch is inside the cube that makes the gingerbread

[8] Objects are not really needed as we may have purely conversational stories but they are likely to be there.

house. I will call those relations – *story relations*. There may also be seen as predicates that describes the story world.

Interestingly, stories in books are never told in terms of base facts but always in terms of relations. At least, I have never read: “And Hansel who was at 120,233,20 with rotation 120,20,0 saw the witch that had appeared at 0,233,20. Hansel changed his velocity from 10,–10,0 to 100,0,0.”

Introduction of story relations will allow the author to specify situations more clearly. For instance, if the author provides relation “near”, he will not need to compare distances of various objects over and over again.

Moreover the previous example of Hansel and the wicked witch is much more readable when using relations: “And Hansel who was approaching the gingerbread house saw the witch in the window. Hansel started to run away.” Although the storytelling framework will not understand English, there will still be places where the author will be able to specify the situations more briefly using relations.

There may also be relations that are inferred from other relations. It has to be allowed but there must not be a cycle in the inference or relations. For instance, we may have relation “actor A facing actor B”, which may be inferred from actors' positions and rotations and base fact “actor A says to actor B” which results in relation “actor A is talking to actor B”.

Thus the *story world* is defined by:

1. the set of story entities - places, objects and actors
2. the set of story facts that describe the entities
3. the set of story relations between entities

The list of story entities, actual values of their story facts and actually valid story relations make the *state of the story world*. The idea of story facts and story relations will be useful later when I will discuss sensing and the perception of the actors.

6.4 Story actions

As we have defined the state of the story world as the set of all places, objects, actors and story facts, it is easy to define *story action* as anything that changes the state of the story world. The behavior is defined as a sequence of story actions then. Typical story actions will be: “run to”, “say”, “perform a gesture”, “follow”. All story actions will surely be parametrized. “run to” action will need a place as an argument, “say” will need a text.

6.5 Story situations and boundary problem

Every situation means a condition. Every condition defines a set of story world states that satisfy the condition. And every such a set *G* has a *boundary* that can be seen as the set of story facts which are similar to at least one of the states from *G* but do not satisfy the condition. It is this boundary where a behavior of an actor will be switched from one to another, which brings two new problems:

1. If these two behaviors are totally different, the result will be funny.
2. If the state of the story world is oscillating on the boundary, the actor will be switching between these two behaviors, which will harm the believability of the actor's character.

For instance, let's say that the wicked witch will pursue everybody she can see and is not too far from her (e. g., their distance from the witch is less then 200m). Here comes a player that has elven boots and can run faster then the witch. Having these elven boots the player may dance on the boundary of these 200m making the fool of the witch. He may watch the witch how she always starts to run towards him and when he runs a bit away and becomes “too far” for the witch, watch her returning to house. Should he be scared by such a witch? He has just discovered the algorithm of her behavior. She is not a witch but dumb computer! This is likely to be the conclusion of the player.

This could be partially solved by giving the author an option to define a second situation when the behavior of an actor should be abandoned. Thus every situation definition will consist of two conditions:

1. First condition expresses a situation when the actor has to begin the behavior.
2. Second condition expresses a situation when the behavior should be abandoned.

Additionally, the author should be able to access the history of executed behaviors and situations that triggered them. Having this history the author will be able to express the situation when a player already tried to approach the house several times and specify a different behavior for this case.

6.6 Story authoring

We have discussed the story world definition in the previous paragraphs. This definitions may be ordered into the sequence of author's tasks that must be done before the story could be executed:

1. The author chooses the virtual environment (e. g., Unreal Tournament 2004).
2. The author creates the concrete
3. The virtual environment will determine to which extent the author may:

   a) define story actions,

   b) define the set of base story facts that may be sensed by the actors.
4. The author defines story relations.
5. The author defines roles by specifying role pair.
6. Finally the author will define the plot.
7. Story may be executed, played and evaluated.

Every level of abstraction lays a basis, which the next layer operates upon. Available story actions are determined by virtual environment. Story relations are inferred from base story facts. Roles are defined using story facts, relations and actions. This is a good sign for the future work as every layer can be solved separately creating a software library and visual tools.

6.7 Chapter conclusion

This chapter has categorized actors' behaviors between interactive and sequential. Virtual actors need to express interactive behaviors when dealing with the player and sequential behavior when performing joint-behavior. Furthermore, the author need to formalize the story world before he may define the actors' roles. The story world is defined as an environment consisting of story entities. Story entities are described with story facts and may be related to each other. Except for entities, the story world also consists of the set of story actions that can be used to change its state.

The introduction of the story facts helped us to define the situation as the condition over the story facts' values and actually valid relations. This brought the boundary problem that should be eased by allowing the author to specify situation that should trigger the behavior and situation when the behavior should be abandoned. Further behavior switching refinement is left to the author to handle by providing reactive behaviors defined by different tools.

How should the authors be supported on their way up to the story execution?

Before describing implemented solution, I would like to state that every level of abstraction could be talked through many times before we will come up with acceptable solution[9] (if such thing is even possible without the experience from higher levels). Therefore the rest of the thesis is derived by informed decisions as well as intuition. Presented storytelling framework is an experiment that probes the presented ground of the story authoring process.

[9] Starting with the presentation of the list of up–to–date closed/partially–opened/open source 3D virtual environment and discussing their pros and cons.

7 Architecture

The thesis moves to the implementation grounds starting with this chapter. It will follow the steps of the story authoring tasks as presented in (ch. 6.6). I will start with the chosen virtual environment and proceed up to the definition of the roles and plot.

7.1 Step 1 – Virtual environment

The storytelling platform is being built over the Pogamut platform that is providing an environment of the Unreal Tournament 2004 *(UT04)*. UT04 is a commercial first-person-shooter game. It allows the player to be present in the environment as an passive observer or one of the actor. UT04 gives a limited way of interaction between player and other actors. Being a first person shooter game, the player may only talk to other character through the console or shoot them. This is a limiting factor for the storytelling application but there is already an ongoing bachelor thesis of Radim Vansa from Charles University at Prague that will address this limitation. Pogamut currently supports maximum of 8 actors. Therefore the storytelling framework will not be suitable for large stories.

7.2 Step 2 – Sensing and acting in the environment

Pogamut is implemented partially in Java and partially in UnrealScript (native language of the UT04). The UnrealScript part of the platform - GameBots2004 *(GB04)* - provides a means for the remote control of UT04 avatars for anybody who implements GB04 textual protocol. The protocol is carried over TCP/IP, which allows the author to run the logic of actors on a different machine then UT04. Figure 10 provides a high–level architectural overview of the Pogamut platform picturing an iteration of sense–reason–act mechanism. There is the UT04 server with GB04 on the left and two bots (Tom and Sheena) inhabiting the virtual world. When GB04 notices that Tom can see Sheena, it generates an event for the Tom's mind and sends it via TCP/IP (1). The event is caught by the Pogamut's library GaviaLib and translated into thr Java object that is presented to the Tom’s agent as an sense event *SeePlayer* (2). The Tom’s reasoning algorithm (that is to be implemented by the user of Pogamut) decides that he should greet Sheena (3) and issues the *SayPrivate* command (4). The command is translated into the GB04 message (5) that is picked up by the GB04 that makes Tom’s avatar to do it (6).

Additionally, Pogamut features a server control connection that allows to observe the environment from the position of the omniscient bodiless agent. This connection could be utilized by the story manager.

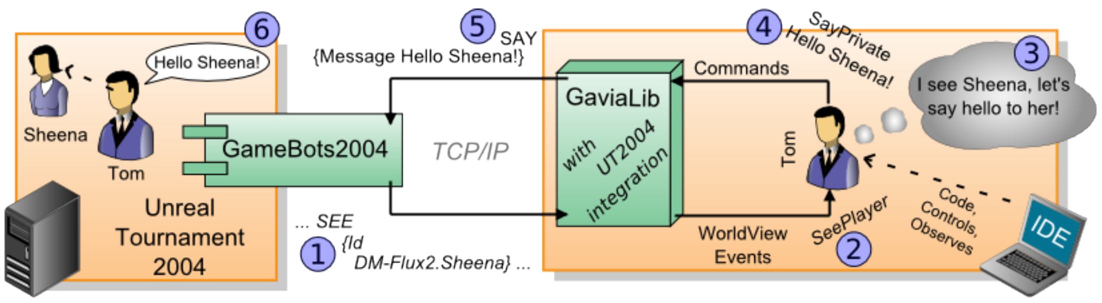


Figure 10 – The high–level architecture of the Pogamut 3 platform

What does the use of the Pogamut imply?

1. The set of the senses is fixed, implied by the UT04.
    - self awareness (location, rotation, velocity), limited actor vision (objects, other avatars, players – no world geometry is included)
2. There is no support for tagging the environment with names of places.
3. The period of the sense updates is fixed.
    - 200 ms
4. The UT04 provides only low-level actor actions.
    - move to (absolute) location, turn to, jump, say
    - no support for remotely controlled skeletal animations
5. GB04 speaks in the terms of events not facts.

How does it affect the storytelling framework?

1. The set of base story facts will be rather limited. For instance, if the author will require to simulate states like hunger, boredom or mood, he will have to provide own simulation mechanism.
2. We will have to provide a mechanism for the definition of the places inside the virtual environment. The places are the only kind of story entities that are not directly supported by UT04.
3. Period of the sense updates dooms our agents to look a bit clumsy. The absence of world geometry information means that avatars will not be able to recognize obstacles in the path of their movement.
4. Absence of high level movement actions means that we will have to define path planning and path following story actions.
5. The storytelling framework needs to take care of the translation of the events into facts.

Although all mentioned points are the matter of implementation, it shows us how choosing concrete virtual environment affects latter story authoring steps.

To look on the Pogamut from the bright side – we do not need to implement the rendering engine, ray casting, collision detection, actor remote control, etc. Additionally, we have an option to seamlessly implement a story manager using server control connection.

Story entities
The UT04 directly supports the recognition of objects and actors inside the environment. The abstraction of places (like house or park) is provided as an additional mechanism by the storytelling framework.

Base story facts
The UT04 allows actors to sense these properties:

1. Absolute location, rotation and the velocity of the actor in the form of triples
   - note that the virtual environment should not need to express those values and may provide only a string identification of the location together with the graph of the location
2. what the actor can see (other actors or items)
3. what the actor can hear

Story actions
The story actions will (again) depends on the chosen virtual environment (UT04) that allows only for low–level move actions and no controlled skeletal animations thus we will not be able to create custom gestures and provide mimics in the latter steps unless we alter the UT04.

7.3 Step 3 – Actor's perception and story relations

To define story relation means to define two things:

1. define the structure of the fact – what it comprises of
2. define the relation's condition

The first point requires the author to decide the structure of the story relation. For instance, if the author is up to define the story relation “near” that will bind two objects and/or actors then story relation will have to consist of two references to objects and/or actors at least. But the author might also decide that there should also be an exact distance incorporated into the relation as it will help him during the definition of the situations' conditions.

The second point is obvious – the framework needs to know how to infer the fact from the current *knowledge base*. The story relations' conditions bring another boundary problem that is similar to the one discussed in (ch 6.5). I have to stress that the framework should allow defining two conditions for each fact: 1) trigger condition and 2) still–valid condition. The former should be used to check whether the relation has appeared. The latter should be used to check whether the fact is still valid. It will allow the author to relax the boundary problem.

Solution for the inferring of the story relations
We are in the situation where we need to evaluate the set of rules – conditions of the story relations – over the knowledge base of story facts. The naive implementation might check each rule against the known facts as long as some rules are firing. This would be clearly too slow. Even if we have a small set of rules we will have to run those rules for each actor to infer story relations from their respective knowledge bases. Thus we should seek different algorithm.

The popular algorithm for the time efficient evaluation of the set of rules (forward chaining) is RETE [Forgy82] that was designed by Charles L. Forgy and published in 1974. The idea of RETE is to create a generalized trie out of the logic expressions. This will create a tree of nodes where every node represents a part of the condition from one rule. Every branch of the tree (path from the root to the leaf) represents one rule. Every new fact that is inserted into the knowledge base is propagated along the tree. If a fact arrives to a terminal node then an appropriate rule will fire.

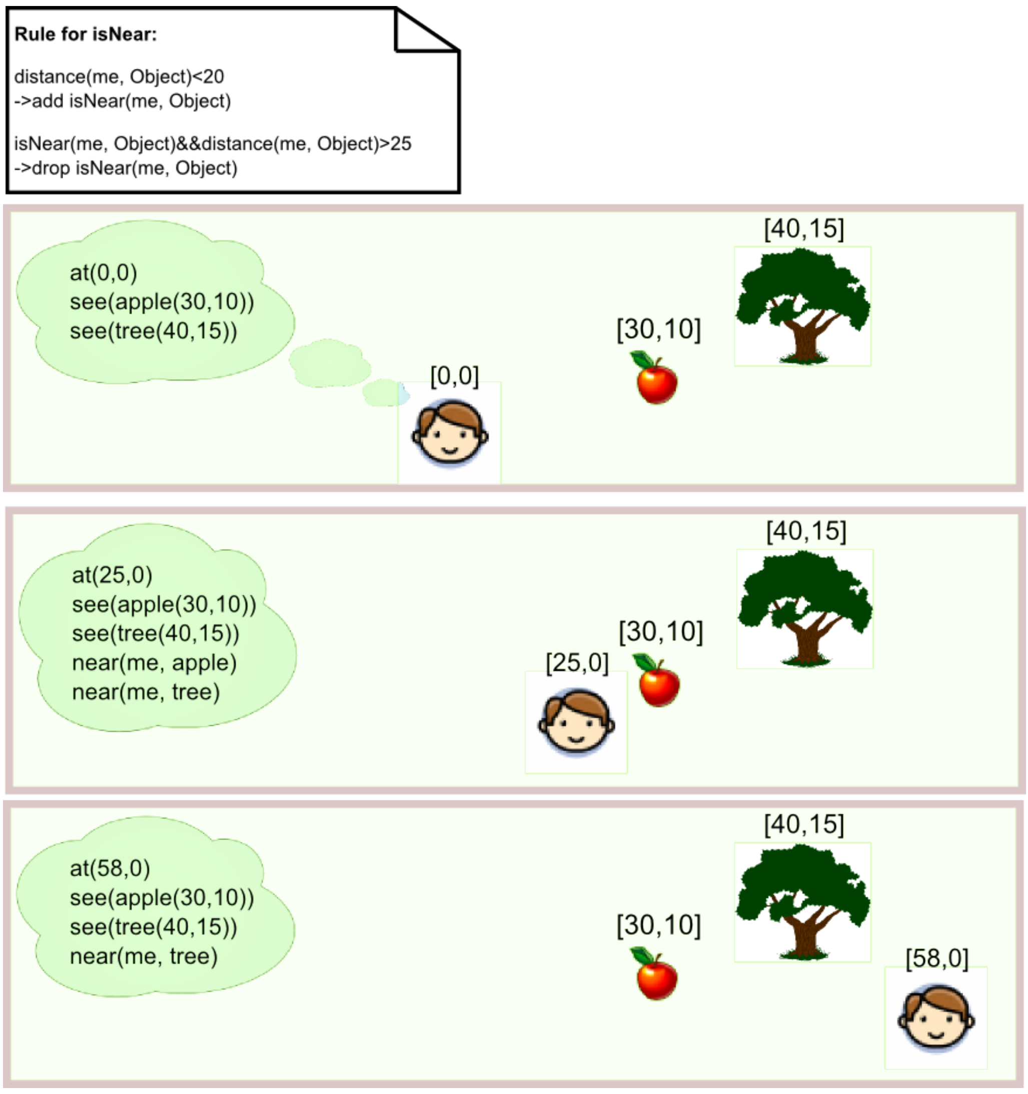


Figure 11 – Example of the story relation "near".

Hammurapi rules

A few existing Java implementation of the RETE algorithm can be found. Before choosing the right library I have to note that we need the implementation to have these three features (see fig. 11):

1. It has to allow inferring new story relations from new base story facts using is-triggered condition of the relation.

2. It has to allow removal of story relations when existing facts are dropped or changes in the way that breaks still-valid condition of the relation.
3. Story relations can not appeared silently inside the knowledge base of RETE as we will be processing them further when they appear or disappear.

We may consider again the story relation “near” that is inferred from an actor named Mr. X and object Weapon. In the beginning, we have only one fact that is describing the location of the Mr. X. As soon as the object Weapon appears in the field of view of Mr. X we require the RETE algorithm to check whether the “near” trigger condition is satisfied. Meanwhile Mr. X will continue walking in direction of the Weapon so the location of the Mr. X is changing and the rule “near” must be rechecked. Whenever the distance to the Weapon is smaller then defined threshold, the RETE algorithm should produce new fact “near”.

Nevertheless, it does not stop here. We also need to be notified by RETE algorithm that the relation should be deleted when the distance becomes greater then the threshold of the still-valid condition.

There exist a few Java implementations of the RETE algorithm. The most well known is the implementation of Drools from the JBoss group, which is unfortunately unusable as it does not support for relations' removal – additional it requires the user to learn another language for condition definition.

Another Java implementation of the RETE algorithm is Hammurapi rules from Hammurapi. It supports all three mentioned requirements. It uses plain Java for the rule definition that is done by subclassing[10] the Hammurapi's Rule class allowing the user to provide arbitrary definition of the rule's condition. I have chosen them because their rule definition is simple and flexible.

7.4 Step 4 – Role definition

The role definition follows the main idea about the virtual acting – more situations the actor will recognize the better acting it will perform.

Situations

The first outlook on situations has presented them as conditions that must be true in order to the situations may be considered as “happening”. But additionally the situation should be also determined by the event that has triggered the recognition of the situation. This is the way the author will think usually. “When the witch saw Hansel eating gingerbread she...”, “When the witch heard that somebody is outside...” Those sentences always begin with sensing some information (an event) from the environment. Every event always happens in some context. “When the witch saw Hansel eating gingerbread she yelled out of the window.” or “when the witch saw Hansel eating gingerbread she start running to him out of the forest.” Context of the former outcome is “the witch is in the house” while the latter is “the witch is in the forest”. The context is nothing else then already mentioned situation condition. Therefore the previous definition of the situation should be extended to be the condition and triggering event.

[10] The term for inheriting a specific class that is used in OOP. The term was introduced by C++ creator Bjarne Stroustrup, who found this term more intuitive than the traditional nomenclature. Result of subclassing a class X is a subclass Y that is a descendant of the class X.

Behavior
The behavior was presented as sequence of intentional actions that are performing some narrative part of the plot. The narrative may require one or more actors. Narratives that require one actor may be expressed in the actor's role while narratives that require more then one actor should be expressed inside story manager (ch 5.4).

7.5 Step 5 – Plot definition

The plot definition is similar to the role definition as it requires specifying situation/behavior pair. The difference is that the plans will be written for the bodiless entity – *story manager* – and have to provide additional features then running story actions:

1. influence other actors by ordering them to do a specific behavior
2. coordinate behaviors of two or more actors

Specific language StorySpeak has been created for the role and plot definition. The language will be discussed in details in (ch. 9, 10).

7.6 Step 6 – Story execution

The story execution is clearly an implementation of the story definition's interpretation. Discussing the inner architecture of the implementation is out of the scope of this thesis.

7.7 Summarization

1. Pogamut is used to control actors inside Unreal Tournament 2004. It limits the story to 8 actors.
2. The base story facts are determined by the environment, they are:
   a) absolute location, rotation and velocity of an actor
   b) information what an actor can see and hear.

   The story places must be additionally defined by the author on the second layer of story abstraction because UT04 does not support tagging of places in the environment.
3. The RETE algorithm, namely its implementation Hammurapi rules, will be used to infer relations between story entities. Those rules will be specified as specific Java classes.
4. The role/plot definition language StorySpeak will be presented in the next two chapters.
5. Story execution is a matter of implementation and should not be discussed further.

8 StorySpeak origin

The rest of the thesis is about StorySpeak language and its interpret. StorySpeak has been developed to support switching between interactive and sequential behavior. Moreover it allows the actors to switch their behavior depending on an observed situation. Situations are expressed as events that happen around the actor together with the context of the event. The idea behind StorySpeak is based on the BDI model of human practical reasoning that was developed by Michale Bratman [Bratman99] more precisely on its formalization AgentSpeak(L) [Rao06].

StorySpeak is based on BDI architecture and can be viewed as an extension to the AgentSpeak(L) language. Firstly, I will first present the BDI idea and give AgentSpeak(L) overview together with its open–source implementation Jason [Bordini06]. Secondly, I will provide a list of features AgentSpeak(L) is lacking to be a language that could be useful to the idea of role / plot definition. Finally, I will show how StorySpeak extends AgentSpeak(L). Chapter 11 will present a few story scripts that will show how StorySpeak could be used to define roles and plots.

8.1 BDI architecture

The BDI architecture is a model of reasoning for implementation of software intelligent agents. It divides agent's mind into three categories: beliefs, desires and intentions. Following explanation of those three categories does not reflect the Bratman's theory completely. It is a traditional interpretation of the BDI architecture for the needs of software intelligent agents.

Beliefs
Beliefs can be viewed as a knowledge base of the agent – it contains every information the agent knows about the environment and itself. Bratman notes that agent's beliefs need not to be true. They represent agent's subjective world-view. I would like to note that StorySpeak does not exploit this and every actor will contain only beliefs that are true.

Desires
Desires express agent's goals. Goals are states of the world the agent wants to reach. The goals may be expressed 1) explicitly by the definition of the state of the world, or 2) implicitly inside agents' intentions.

Intentions
Intentions express agent's ways how to satisfy its desires.

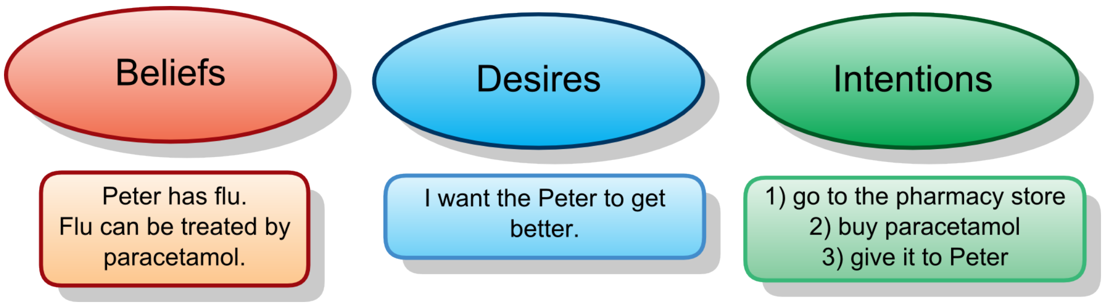

Beliefs
Desires
Intentions
Peter has flu.
Flu can be treated by paracetamol.
I want the Peter to get better.
1) go to the pharmacy store
2) buy paracetamol
3) give it to Peter

Figure 12 – BDI model of reasoning.

The BDI architecture is traditionally extended with plans (BDI+P), which are sequence of actions or another intentions that leads to satisfaction of the desire. The agent may have different desires, which may even be conflicting with each other. Therefore the agent contains also the intention–selection mechanism that chooses which intention should be executed.

Plans may contain not only actions but also references to other plans. They usually contain 1) preconditions and 2) during-conditions that must be satisfied 1) prior the execution of the plan and 2) during the execution of the plan. Hierarchy of plans and plans' condition is making BDI+P architecture quite similar to HTN planning (Hierarchical task network) planning [Silva04].

The BDI+P alone is not a software framework and needs further formalization. Today there exists several such formalization[11] for example JAM, 3APL and AgentSpeak(L).

8.2 AgentSpeak(L)

AgentSpeak(L) is a programming language based on a restricted first–order language with events and actions. A behavior of an agent is expressed as AgentSpeak(L) programs. The language can be viewed as a formalization of the BDI architecture and allows agent programs to be written and interpreted in a manner similar to that of horn–clause logic programs. It was invented by Anand S. Rao and he has shown how to perform derivations in its logic in AgentSpeak(L) original paper. The paper starts with giving a formal definition of the language but it is soon clear that the language can be thought of as an extension of the logic programming. The belief base of the agent is the set of ground (first–order) atomic formula. The AgentSpeak(L) program represents plans how to satisfy agent's desires. Every plan represents a way how to respond to events that are happening around an agent. When a plan is selected for the execution, we say that it has been instantiated. Instantiated plans represent agent's intentions.

```
+location(waste,X)                                … Head
     :    location(robot,X) & location(bin,Y)    … Context
     <-   pick(waste);                            … Body      (1)
          !location(robot,Y);                                 (2)
          drop(waste).                                        (3)
```

Figure 13 – Example plan from the Rao's paper that is about waste disposing robot.

[11] http://en.wikipedia.org/wiki/BDI_software_agent [16. 4. 2009]

Every plan consists of three parts: *head*, *context* and *body*. The head of the plan is formed by the *event* that consists of *triggering symbol* (addition “+” or deletion “–”) and *event term* (the Prolog term). The agent receives events in form of +terms from the environment that are matched against event terms from addition plans head. Whenever an unification (mgu) of the event and head exists, the plan's *context* is checked whether it holds in agent's belief base. If so then the plan is instantiated as an intention.

We may interpret example plan (see fig. 13) as follows: If waste is spotted at location X (head) and the location of robot is the same and location of garbage bin is Y (context), then pickup the waste (body 1, atomic action), execute the plan `location(robot, Y)` (body 2, supposed to move the robot to location Y) and drop the waste (body 3, atomic action).

Rao then compares the AgentSpeak(L) language with logic programming[12]:

In summary, a designer specifies an agent by writing a set of base beliefs and a set of plans. This is similar to a logic programming specification of facts and rules. However, some of the major differences between a logic program and an agent program are as follows:

- *In a pure logic program there is no difference between a goal in the body of a rule and the head of a rule. In an agent program the head consists of a triggering event, rather than a goal. This allows for a more expressive invocation of plans by allowing both data–directed (using addition/deletion of beliefs) and goal–directed (using addition/deletion of goals) invocations.*
- *Rules in a pure logic program are not context–sensitive as plans.*
- *Rules execute successfully returning a binding for unbound variables; however, execution of plans generates a sequence of ground actions that affect the environment.*
- *While a goal is being queried the execution of that query cannot be interrupted in a logic program. However, the plans in an agent program can be interrupted.*

AgentSpeak(L) interpretation

Finally, Rao provides formal operational semantics for the language. For the sake of brevity I will only provide an AgentSpeak(L) interpretation diagram (see fig. 14). Through out the following text I will refer to the language interpretation as the *agent's reasoning*.

[12] Following text is cited from [Rao96].

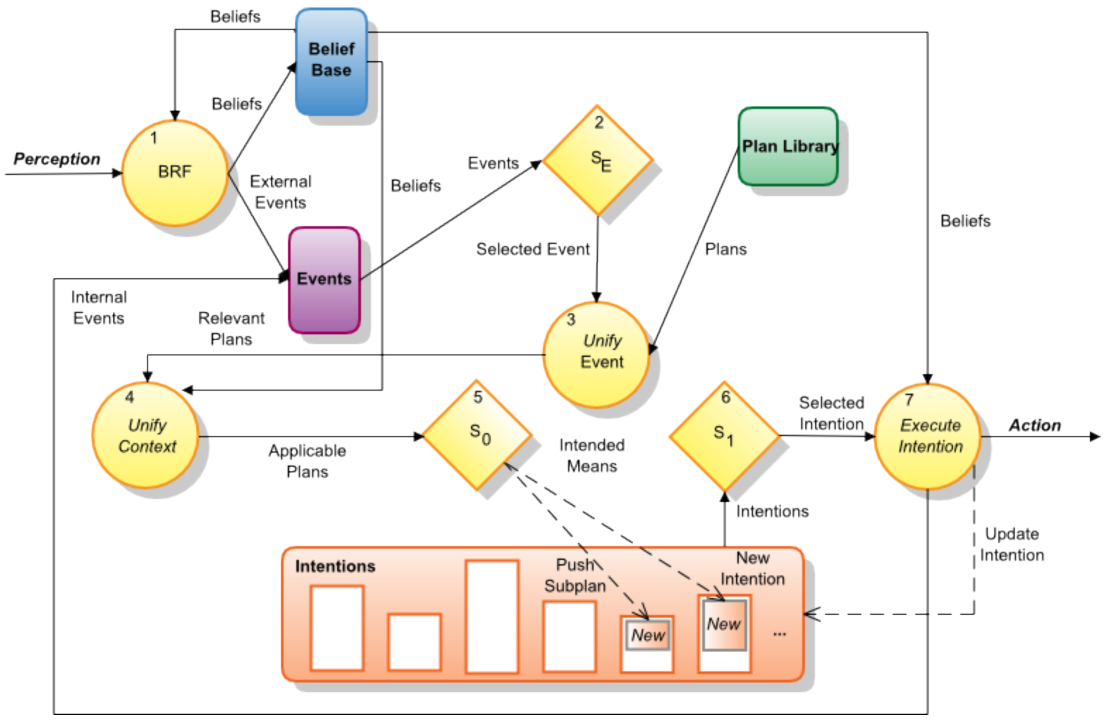


Figure 14[13] – Reasoning algorithm of the AgentSpeak(L) agent

Whenever an event (in the form of ground term) arrives via agent's perception, it is stored within the list of events and the belief base is updated of this event. The event processing is done during the reasoning of the agent. Events are internal or external. External events are all events that are sensed from the environment. Internal events are events produced during the reasoning of an agent. The reasoning algorithm works as follows:

1. The event *E* is selected from the event list using event selecting function S_e.
2. The plan library is searched for all plans, which head unifies with the event *E,* creating a list of possible plans L_P.
3. Context of the plans from L_P is checked. Plans which context holds are put into the list of applicable plans L_A
4. Option selection function S_o selects one plan *P* from the list L_A and: either
   a) If the event *E* is external, new intention is added to the intention list
   b) If the event *E* is internal event from the intention *I*, the plan *P* is added on top of the actions from intention *I.*
5. The intention selection function S_i selects one of the intention to execute, the action may be:
   a) belief base change
   b) belief base check
      - if belief base check fails, the intention fails (the intention is removed from the list) and an deletion event *–term(E)*, where E is the head of the intention's plan, is added to the event list
   c) agent atomic action[14]

[13]The figure is based on one the figure from [Bordini01].

[14] Atomic from AgentSpeak(L) point of view.

d) plan call
   - produces an internal event

Note that the Rao's paper does not provide concrete mechanism for the case when there is no plan for deletion event. The concrete implementation is further provided by Jason.

It is clear that the program written in AgentSpeak(L) will produce actions as long as it is fed by events it has plans for.

We can see from the fig. 14, that AgentSpeak(L) uses three functions during the reasoning algorithm, S_e, S_o and S_i without specifying them further or giving hints what the specification could be.

Plans as situation/behavior pair

The event–context-plan idea of the AgentSpeak(L) language is similar to the situation / behavior pair from (ch. 5.3). Events are informing the agent about changes in the environment allowing the agent to match a new situation according to his plans. The situation is defined by the plan's head and the context. The plans' bodies represent actors' behaviors to perform.

The plans in AgentSpeak(L) are interruptible. It will allow to switch between interactive and sequential behavior without destroying the intention of the actor.

8.3 AgentSpeak(L) extensions

Although Rao's AgentSpeak(L) is a fine formalism of the BDI architecture, it is far from being useful as an actor's role definition language (and any agent in general) as it lacks several things:

1. The formalism speaks only in Prolog terms, which really limits the context definition of the plan. The author of the story should be given a way to perform more checks (e.g. `distance(X) <  100`) inside the context definition.

2. Even though the instantiated plans *(intentions)* are interruptible it does not offer any mechanism for checking the context again when the execution returns to the interrupted plan. There are three states the resumed intention may be in:
   a) The desire has been meanwhile satisfied and there is no need to continue the execution of the intention.
   b) The situation around the agent changed in a way, which prevents the execution of the intention. Let us consider the plan for the waste disposal. If the plan is interrupted right after the second call (see fig. 13, body row 2) with intention that drives the robot from the location Y, then returning to the plan will prove fatal. The waste disposal plan will execute `drop(waste)` at different place then Y.
   c) The situation around the agent is still valid for the behavior to continue its execution.

   AgentSpeak(L) always assumes the third option.

3. The only way to specify plans' priorities is through the means of intention selection function S_i. Unfortunately, the S_i is not given additional information about the plan's priority. The grammar of the language should be extended for the author to specify a plan's priority.

4. AgentSpeak(L) does not allow to write plans for events in the context of executed intention explicitly. We can not write plans that may be instantiated only during the execution of certain plan. This would allow for finer control over the events that are passed to an agent. This extension would allow us to write *subplans* handling appearance of new waste while the agent is in the middle of example plan.

5. There is no way to express plans' timeouts.

6. There is no mechanism for providing a plan that should be executed whenever there is no intention inside intention list and no events inside event list. This will allow definition of a default actor's behavior.

7. The expressions that are allowed inside the body should be extended too, at least to allow simple if–then–else check; otherwise we would have to write actions or plans (that can perform if–then–else check via context) for every situation whenever a simple decision is needed.

8. Plans can not directly return values, which prevent creating plans subroutines with decision points.

StorySpeak implementation contains many extensions to original AgentSpeak(L) solving those problems plus providing additional features. Through out the following text, I would refer to these points as AgentSpeak(L) extensions *(ASLe)* together with specific number.

Jason – Java implementation of AgentSpeak(L)
Rather straightforward implementation of AgentSpeak(L) is Jason. It is described in the paper [Bordini01]. Jason provides a Prolog implementation for the belief base and provides a Java bindings[15] for finer control over executed intentions and plans that should be instantiated. The plan may be annotated with key/value pair hat may serve as the basis for the definition of the selection methods. These annotations makes the basis for the implementation of the selection functions. This gives the user a way to control the selection functions.

Also Jason is defining the deletion event processing in very sensible way. When the action fails, Jason does not remove the whole intention from the intention list as AgentSpeak(L). Instead, only the actions from the top instantiated plan *P* is removed from the intention and deletion event *-term(P)* is generated. If the deletion event is chosen to be processed and no applicable plan is found, then next top plan in the *intention(P)* fails. This mechanism is quite similar to the exception handling from Java.

[15] The user has to subclass the Jason's agent to provide a specific implementation for the selection functions.

Other Java BDI implementations
Other popular BDI implementations in Java are JACK and JAM.

JACK is an industrial platform therefore it is closed–source. Even though the JACK is also based upon BDI idea it is more suitable for multi–agent system according to Wooldridge as the system is based upon the negotiation between respective JACK agents. Thus it would allow only for author–centric approach where the JACK agent would be a story manager that controls all the actors at once otherwise writing joint–behaviors will be cumbersome.

The JAM [Huber99] language was created for the control of mobile robots. It contains while cycle but it has a rather ugly syntax that is mixing prefix and infix notation.

9 StorySpeak

StorySpeak is based upon the idea of AgentSpeak(L) but provides a different grammar and extended semantics for its language. It is incompatible with AgentSpeak(L) or its Jason implementation. StorySpeak is the name of the language as well as the interpret implementation in Java. It was designed to support all mentioned AgentSpeak(L) extensions *(ASLe)* from (ch. 8.3). It also implements ideas presented in (ch. 5.5). The reasoning cycle of StorySpeak agent is very similar to the one from fig. 14, the differences are:

1. Functions S_e, S_o and S_i have fixed implementation but customizable through the plan annotations (plan annotations idea is borrowed from Jason).
2. StorySpeak agent does not have an intention list but the intention trees and S_i picks the plan with the highest priority from all leaf plans among intentions (will be explained later).
3. StorySpeak does not evaluate only top action from the plan it executes, but it executes the whole batch of actions – how long the batch would be is up to the author. This provides a flexible way how to express atomic operations inside plans.

9.1 Additional StorySpeak extensions

Except for ASLes (the list from ch. 8.3 may serve as additional list of StorySpeak features), StorySpeak provides extensions that allows the author to provide both interactive and sequential actors' behaviors. StorySpeak implements these additional extensions *(SPes)*:

1. The possibility to write so–called *template plans*.

   Template plan is a plan that should be executed by more then one agent together allowing the author to write coordinated joint–behavior plans.

2. Issuing plan delegation in parallels that allows the author to control actors directly from the story manager plans.

Plan delegation allows the story manager to order an actor to execute plan that is defined in the story manager library. This feature is also needed during the definition of a joint-behavior plan.

3. Tight bindings to Java language.

   The interpreter may work with any class or object directly within StorySpeak program[16], which enables endless extension of the language by Java libraries. StorySpeak also allows writing Java expressions inside the plan context or body[17].

4. Framework for writing *Java Prolog beans*.

   Java Prolog beans are automatically translated into Prolog terms, may be matched by Prolog unification and are automatically translated back into beans if needed. Additionally any object may be part of the Prolog term via string translation allowing Prolog terms to store references to concrete Java objects. This feature allows the author to easily express story relations as simple Java classes.

5. Usage of tuProlog[18] for the belief base.

   StorySpeak uses tuProlog as a simple database engines. It does not exploit the use of Prolog predicates. Also StorySpeak does not use Prolog lists at all.

Rather then to formally define a lot of StorySpeak terms, I will present all StorySpeak features that target the ASLes (ch 8.3) and SPes together with their informal semantics. This presentation will be followed by the explanation of StorySpeak interpretation algorithm. The following syntax definition[19] is only a subset from the complete StorySpeak language syntax. The complete syntax can be found in Appendix B.

The following text is using "Prolog term" in a restricted sense. It may be any Prolog term except for the list.

9.2 StorySpeak plans and basic expressions

Every StorySpeak actor contains own library of plans. The notion of StorySpeak plan is to provide a way for specification of one situation/behavior pair.

Every StorySpeak actor file is a list of plans separated with `#`. Every plan must have a head and a body – a context may be omitted. A head consists of a triggering symbol `+` or `-` as in AgentSpeak(L) followed by `!` and a Prolog term that is unified with incoming events. If mgu exists the plan is *relevant* for the event. The body consists of batches of expressions. Batches are separated with `;` and expressions inside a batch with `,`. The batch plays a role of the atomic operation from the StorySpeak point of view, unless it is interrupted by `plan call`.

[16] Via Java Reflection API.
[17] Everything except bit operations, array access operators, operator *new* and class names.
[18] Open–source (LGPL) implementation of Prolog in pure Java . http://www.alice.unibo.it/xwiki/bin/view/Tuprolog/ [16. 4. 2009]
[19] Syntax definition is written in EBNF.

Plan syntax definition:[20]

```
plan = ('+'|'–')'!'prolog_term              … head
          [ ':'  logic_expression ]        … context
            '<-' plan_body                 … body
       '#'

plan_body = expression (',' expression)*
       (';' expression (',' expression)*)* [';']

logic_expression = …
```

> The logic expression syntax is similar to the syntax of Java expressions (using &&, ||, ? :, brackets, etc.) additionally the author may use belief checks (`?prolog_term`) inside them.

Every StorySpeak actor maintains a plan library where the plans are sorted according to their *relevance* (specified via plan annotations, ch 9.9). When an event is processed, StorySpeak searches for an *applicable plan* that has the highest relevance (which defines S_o function from AgentSpeak(L)). The plan is applicable if it is addition plan, the head of the plan unifies with the event (mgu) and the plan's context holds (evaluates to `true`). If the context of the plan is missing StorySpeak assumes it evaluates to `true`. The plan's head and context specifies the situation, while the body defines the behavior.

Deletion events are handled differently in contrast to AgentSpeak(L). When some plan fails, StorySpeak does not produce a deletion event but immediately searches the plan library for applicable deletion plan. If such plan is not found, the failure is propagated as in Jason (ch 8.2).

In fact, the algorithm that chooses applicable plan for the event is more complicated then iterating through the agent's plan library and will be explained later in (ch. 9.11).

First few StorySpeak `expressions` (those which are present also in AgentSpeak(L), see ch. 8.3):

1. `belief checks = '?'prolog_term`

   Belief checks perform Prolog queries over the belief base of the actor. Provided term may contain unbound or already bound variables (begins with upper–case letter) or anonymous variables _. If belief check fails, it will not trigger failure of the plan (as in AgentSpeak(L)). It will return a boolean value instead that may be tested with ternary operator `? :` and `fail()` method may be used to trigger the failure of the plan.

2. `plan call = '('priority')''!'prolog_term`

   This will try to instantiate a new plan as a child of the current one (explained later). If no suitable plan is found, the caller plan will fail. The plan's priority may be specified.

[20] Following syntax definition of the plan is partial, the syntax for the plan is richer and allows to define conditions to be checked during the execution or conditions for early success of the plan.

3. `method calls = method_name'(' arguments ')'`

   StorySpeak language may be extended with arbitrary number of methods. These methods may be used for anything the user wants and may contain variable number of arguments. They can not be overloaded. Method names always begin with lower–case letter and they are implemented in Java. Writing a new StorySpeak method is as simple as subclassing specific class within Java.

4. `belief base change = ('+'|'-')prolog_term`

   Belief base change is used to alter the belief base. The author may assertz[21] or retracts a fact from the belief base of the actor or story manager. Moreover, Appendix B is containing a syntax that allows to change the belief base of any StorySpeak actor.

Thus we may rewrite an example plan from the Rao's paper as follows:

```
+!location(waste,X)                                    … Head
     :    ?location(robot,X) && ?location(bin,Y) … Context
     <-                                                … Body
          pick("waste"),                               (1)
          !location(robot,Y),                          (2)
          drop("waste");                               (3)
#
```

Figure 15 – Rewritten example plan from [Rao96] assuming that user has specified methods pick/1 and drop/1. Quotes around the `waste` are notable. StorySpeak forbids using Prolog atoms anywhere else then inside Prolog terms. Also notice that logic conjunction is using && as in Java.

9.3 Additional expressions

Apart from basic expressions that was present in the original AgentSpeak(L), StorySpeak allows additional types of expressions.

1. `assignment = variable_name'='expression`

   This is used to assign a value to a variable – keywords `null`, `unbound` or `self` may be used to assign null value, unbound the variable or gain a reference to the executor of the plan (note that with plan delegation an executor does not need to be the same as the owner of the plan), keywords – `true`, `false` may be used to assign logic values. All variables must begin with an upper–case letter.

2.
   ```
   if-then-else = expression '?' then_expression
                                [ ':' else_expression ]
   ```

   Sometimes, it is needed to assign a value based on a logic expression or check whether some variable has been already bound or not – therefore there exists ternary operator `? :`. This operator may be used in the context as well. It may also be nested inside then or else branch. The else branch is optional. Additionally, when this ternary operator is part of the plan body, it is allowed to insert batches of expressions into branches. The batches have to be wrapped with curly brackets.

[21] The term assertz is used to express a fact that a new term is added to the end of the belief base .

```
3. full set of binary and unary operators
   relation          <, >, ==, !=
   logic             &&, ||, !
   arithmetic        +, -, *, /, %, ++, --
   assignment        +=, -=, *=, /=
```

Note that StorySpeak features lazy evaluation of logic expression and that the operator precedence is the same as in Java.

```
+!see(person(Name))                                    (1)
     :    ?greeted(Name, Time) ? Time < time()-10 : true (2)
     <-   say("Hello " + Name),                        (3)
          Time ? -greeted(Name, Time),                 (4)
          +greeted(Name, time());                      (5)
#
```

Figure 16 – plan with more complex context and body

The figure 16 deserves a bit of explanation. The plan defines a behavior for greeting people and takes special care not to greet them too often. The head (1) assumes that agent may receive an event `see` with the object as its first argument – successful unification with a `see` event bounds the `Name` variable. The context (2) will check, whether we have not already `greeted` the person. If so, it will check when (method `time()` returns current time in seconds). If the actor does not greet person yet, it will return `true`. (3) will say aloud the greeting (concatenating the strings). (4) will check whether the variable `Time` is bound (note that unbound variable evaluate to `false`), if so, retracts the fact from the belief base. (5) assertz information about greeting the person `Name` at `time()` to the belief base.

The context in the figure 15 makes an example for the ASLe 1 and shows that StorySpeak is more flexible then AgentSpeak(L) or Jason.

9.4 Variables and unification

StorySpeak is working with Prolog terms therefore it is natural to express variables as any identifier starting with upper–case symbol. StorySpeak is interpreted language and one of its aims is to be as brief as possible. That is why there is no need to declare variables before hand[22] and first usage of the variable creates a variable in current execution context with value *unbound*. Unbound value means that the variable will be evaluated into a free variable during the unification. Once bound, the variable will represent a specific value inside Prolog terms as well as in other expressions. The author may use a keyword `unbound` to make the variable free again.

[22] This makes the language to be type–unsafe.

```
+! eatLunch(Food)
<-    ?at("fridge", Location),                              (1)
      goTo(Location),                                       (2)
      open("fridge"),                                       (3)
      take("fridge", Food),                                 (4)
      Location = unbound,                                   (5)
      ?at("table", Location),                               (6)
      goTo(Location),                                       (7)
      ?see(Chair)                                           (8)
      sitdown(Chair),                                       (9)
      eat(Food);                                            (10)
#
```

Figure 17 – example of the plan with belief base queries. Terms beginning with `?` are belief checks (or queries). Belief query at (1) bounds the variable `Location` and if we did not unbound the variable at (5) the belief query at (6) would fail and the agent won't move to the `table` with method `goTo` at (7). Of course the user may use another variable, e. g., `Location2`.

9.5 Calling Java methods, accessing Java fields

One of the feature of StorySpeak is the utilization of Java Reflection API to invoke methods on the objects and accessing their public fields dynamically. The grammar is the same as in Java.

```
1. accessing field = Variable'.'fieldName |
                     '('expression')''.'fieldName

2. java method call =
      Variable'.'methodName'('args')' |
      '('expression')''.'methodName'('args')'
```

Additionally, StorySpeak provides a simple access to getters – if field is not found on the Java object a getter with the same name is tried. For instance, `self.location` refers to a field `location` of the executor of the plan or (if the field `location` is not present) to a method `getLocation()` of the executor. The user may chain those calls, e. g., it is possible to write `self.actions.say("Hello")`.

Where StorySpeak is lacking is 1) accessing static methods on classes, 2) working with arrays and 3) instantiation of new objects. The first problem can be overcome by inserting a global variable into StorySpeak with a value containing Java class[23]. Second problem can be solved by using Java collections and third one (if really needed) may be solved by creating a new StorySpeak method that would invoke constructors.

Presented feature is not available in the Jason language, where the user is forced to define Jason's methods for such cases.

[23] There is already present a global variable System that refers to the System class from Java.

9.6 Template plans, plan delegation and parallel expressions

Template plans introduce the notion of *master/slave* actors. The *template plan* always contains an executor actor *(master)* and contains a list of *slave* actors. If a slave actor is about to execute the *template plan,* it will not do anything and let the *master* actor to control him. This control is in terms of belief base querying and alteration, executing story actions or ordering the actor to execute some other plans. Together with the ability to delegate execution of a certain plan to a slave actor, the template plans may be used to write coordinated joint–behaviors.

Additionally, StorySpeak allows user to write parallel expressions that are crucial in order to write joint–behavior plans. The author would not be able to specify actions that should be executed in parallel without them.

```
1. parallel behavior =
     '||' '(' expression (',' expression)*
            (';' expression (',' expression)*)*
         ')'
```

Parallel behavior is expressed inside brackets prefixed with two vertical bars. Batches of expressions inside those brackets will be executed in parallel. Batches are separated with `;` and actions are separated with `,`.

```
2. plan delegation = Actor !! prolog_term
```

The plan delegation differs from ordinary plan call with additional ! and variable name that should contain an instance of an actor. Every actor in StorySpeak has its unique name and is accessible via `actor(name)` method.

```
3. template plan =
      (+|-)!
      [ variable (, variable)* ]            … template
      prolog_term
            [ :logic_expression ]
            <- plan_body
      #
```

Template plan definition differs from the ordinary plan by the `template` part inside square brackets where the names of the slave agents are expressed. Slave agents will be available in the template plan body under variables that are defined in `template`.

```
4. template plan call =
      ! [ expression (, expression)* ] prolog_term
```

Similarly to the template plan definition, we have to specify slave actors during the template plan call. Expression must evaluate to an actor object. There also exists a template plan delegation variant with `!!`.

```
+![Agent1, Agent2] moveCouch(Couch, Room)
<-
      Parallel = ||(
            Agent1!!goTo(Couch.location),
            Agent1!!grab(Couch);

            Agent2!!goTo(Couch.location),
          Agent2!!grab(Couch)
        ),
      Parallel.waitAll(),
      Parallel = ||(
          Agent1!!move(Room);
          Agent2!!move(Room)
        ),
      Parallel.waitAll(),
      ||( [Agent1]!!drop(Couch), [Agent2]!!drop(Couch) );
#
```

Figure 18 – Example of the joint–behavior plan to take the couch and move it to another room. Expressions inside ||`( ... )` are done in parallel and the master agent waits on the completion of the inner expressions at the next row. The semicolon separates batches. `!!` stand for plan delegation and the term after `!!` is plan of the master or respective slave agents. StorySpeak is utilizing its bindings to Java. The `waitAll()` is public Java method of the parallel handle class from Java.

9.7 Failing plans

StorySpeak contains two types of plans (as is the case of AgentSpeak(L)). *Addition plans* beginning with `'+'` and *deletion plan* beginning with `'-'`. Whenever an event is being processed by StorySpeak or the plan call is being evaluated inside the body, StorySpeak will search for appropriate plan among *addition plans*. Whenever a plan fails (due to the failure of atomic action or called plan, or method `fail()`), StorySpeak searches for the *deletion plan* to execute instead of the plan that has failed. This mechanism may be used to provide an alternative plan when the original plan fails. If the *deletion plan* fails, the failure is propagated to the plan that originally called the one that has failed. If no such plan exists then the failed plan was the first instantiated plan of the intention and the whole intention is dropped.

```
+!eatLunch(Food)
      :     Food == "apple"
      <-
            ?at("fridge", Location),                        (1)
            goTo(Location),                                 (2)
            open("fridge"),                                 (3)
            take("fridge", Food),                           (4)
            close("fridge"),                                (5)
            eat(Food);                                      (6)
#

-!eatLunch(Food)
      :     Food == "apple" && at(self, "fridge") &&
            ?opened("fridge") && ?inside("fridge", "orange") (7)
      <-    take("fridge", "orange"),                       (8)
            close("fridge"),                                (9)
            eat("orange");                                 (10)
#
```

Figure 19 – Example of the *deletion plan* that provides an alternative for the plan `eatLunch("apple")`. Let's say that the action (4) fails because there is no apple inside the fridge. This failure will cause the plan to fail and StorySpeak will try to

search for the *deletion plan* with the same head (or more precisely it will try to find *deletion plan* with the head that unifies with the original event that triggered the *addition plan* that has just failed). Ultimately StorySpeak finds the deletion plan as defined above, checks its context and instantiate it.

Note that the *deletion plans* could be used for the endless execution of the specific plan creating plan: `-!plan() <- !plan() #`.

9.8 Subplans and handling of events inside instantiated plan

StorySpeak allows writing plans that may be instantiated only in the context of the execution of specific plan. These plans are called *subplans*. Every subplan has an *owner plan*.

Extended plan syntax:

```
plan = ('+'|'-')'!'prolog_term
          [ ':'logic_expression ]
          '<-' plan_body
          [    '{'
                    (plan)+              … subplans
               '}' ]
     '#'
```

Every subplan may be executed only if the owner plan is currently being executed by the agent. Thus we have a mechanism for creating different behaviors in the context of executed plans. This allows for finer control over the executed plans.

9.9 Plan annotations

Every plan may contain an arbitrary number of annotations. Annotations are defined right after the plan head inside square brackets. Every defined annotation is also present as a variable during the plan body execution. There exists a few annotations with specific meaning such as `Relevance`. `Relevance` annotations order the plans inside an actor's plan library. If more then one plan is applicable, the one with the highest relevance is instantiated. Another notable annotation is `Timeout` that may be used to limit the maximum execution time the plan is given (in seconds).

```
plan = ('+'|'-')'!'prolog_term
     '[' annotation (',' annotation)* ']'  … annotations
          [ ':' logic_expression ]
          '<-' plan_body
     '#'
annotation = variable '=' expression
```

Note that all annotation names must start with an upper–case letter too and that they are present as variables in the execution context of the plan.

```
+!see(person(Name)) [ Relevance = 10 ]
      :     at("kitchen")
      <-    say("Hello " + Name + "! Are you hunting the fridge
            again?");
#
```

```
+!see(person(Name)) [ Relevance = 20 ]
      :     at("kitchen") && state("hungry")
      <-    say("Hello " + Name + ", are you hungry as much as
            me?");
#
```

Figure 20 – Example plans with `Relevance` annotation. When the agent is in the kitchen, is hungry and encounters a person, context of both plans is true. StorySpeak will use the value from `Relevance` annotation in such cases and instantiate the plan that has the highest relevance. If `Relevance` annotation is not present, a plan receives default relevance of 1000.

9.10 Definitions

Now I will informally define a several terms in order to be able to present a reasoning algorithm of StorySpeak.

Plans
Plans are of two types: *standard* and *template plans* – both of them may be referred simply as *plans*. Every plan has its *executor* and in the case of template plan also subordinates (or *slaves*). Also every plan is either an *addition plan* (prefix `'+'`) or *failing plan* (prefix `'-'`) and consists of a *head*, a *context* and a *body* and may have *during conditions* and *early success conditions* specified. Additionally, every plan may have *subplans*. Every *subplan* has an *owner plan*. If a plan is present in any plan node, it is called *instantiated plan*. If the plan still contains actions to execute, it is called *unfinished plan*. If the plan has no actions to execute it is called *finished plan*.

Events
Every event that is passed into StorySpeak contains a *Prolog term* and possibly a *priority*.

Intention trees
Intentions in StorySpeak are quite different from the concept of AgentSpeak(L). Intentions in AgentSpeak(L) are list of actions, which resides in the *intentions list* – even though this is partial true for StorySpeak as well (we have to store the plan's actions somewhere). Nevertheless, instantiated plan's actions are not stored in an intention, but they are wrapped by the *plan node,* which resides in the *intention tree*. Root of the intention tree corresponds to the agent's desire as it contains a plan that was instantiated according to some event.

Why can not we do without intention trees? After all, actor will need to maintain the list of plans, it wants to execute, sorted according to their priority, which can be viewed as priority queue? That is because we need to track two things:

1. Whenever a plan fails we have to propagate the failure of the plan down the intention tree in search for a *failing plan*. Thus every plan node must maintain a reference to its caller (except the root of the intention tree).
2. Every plan may contain a *plan delegation* expression, thus every plan may have more children and it needs to store their references in case it fails so it will be able propagate failure to its children.

StorySpeak recognizes two types of *links* (edges between nodes) inside intention trees – *strong* and *weak*. Child that is connected to a parent with a strong link is called *strong child*. Child that is connected to a parent with a weak link is called *weak child*.

We need to differentiate between three types of nodes to track which nodes are subjects for executions:

1. *strong leafs*
   - nodes that have no children at all
2. *weak leafs*
   - nodes that have no strong child and at least one weak child
3. *plan node*
   - nodes that have at least one strong child

When an actor performs a reasoning it chooses a *leaf plan* with the highest priority, where the agent is an executor or the slave, and executes its next batch of actions. We can not execute plan nodes as they have issued a plan call and we have to wait for its strong child to complete (or fail).

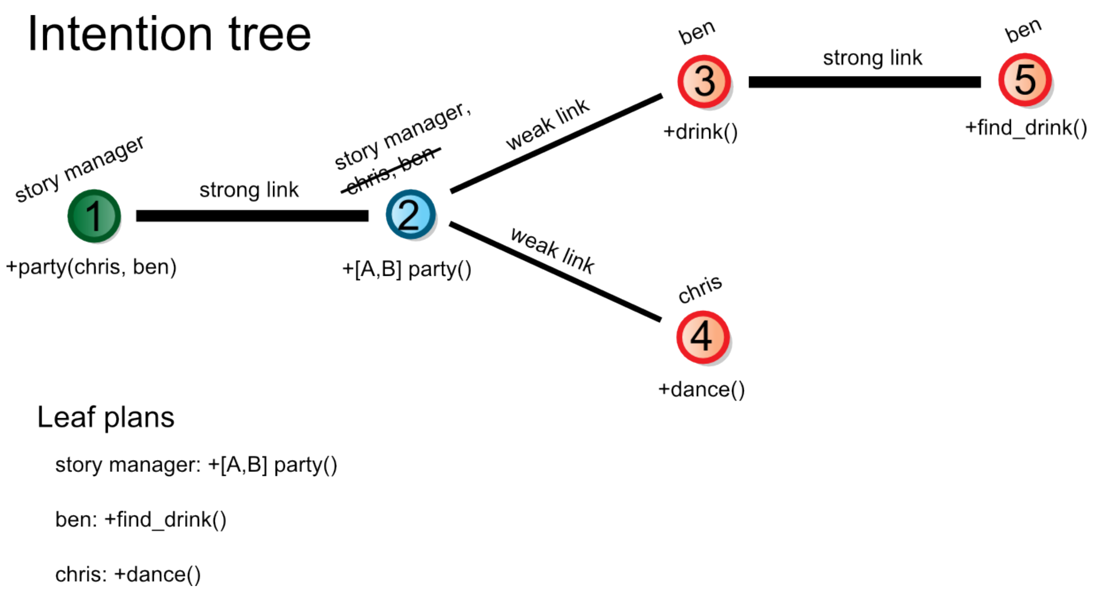


Figure 21 – Example of the story manager intention tree "party" (1). The intention tree is describing an intention tree of the story manager, that should order actors *chris* and *ben* to party. The story manager first executes template plan `+[A,B] party()` (2) bounding *chris* as `A` and *ben* as `B`. Then the story manager orders *ben* to drink (3) and *chris* to dance (4). because *ben* does not holding any drink then he sets out to find one (5). The story manager has delegated plans `+drink()` and

`+dance()` to *ben* and *chris* respectively. Therefore *ben* and *chris* has been removed from actors that are executing the `+[A,B] party()`. Plans 1 and 3 are nodes. Plan 2 is a weak child and plans 4 and 5 are strong children.

Actor
Every StorySpeak actor has:

1. a *plan library P_L*
   - Plan library is made of list of plans that is sorted according to the plans' relevancies.
2. a *forest of intention trees F_I*
3. list of leaf plans L_L sorted according to the priorities
4. a *list of events E*
   - All incoming events between the reasoning iterations are stored here.

9.11 Interpretation algorithm

An interpretation algorithm is periodically called by the storytelling framework for every actor that is present in the story world. The framework is multi-threaded and uses ScheduledThreadPoolExecutor.

```
reasoning():
      Ec = copy(E)
      E = 0
      while Ec != 0:                              (1)
            process_event(front(Ec))
            pop(Ec)
      if peek(LL) != 0:                           (2)
            Plan = peek(LL)
            execute(Plan)
      else:
            process_event(noPlan_event)           (3)
            if peek(LL) != 0:
                  Plan = peek(LL)
                  execute(Plan)
end
```

The reasoning method does three things:

1. it processes all events that have come since the last execution of the algorithm,
2. it executes a next batch of actions from the leaf plan with the highest priority where the agent is an executor or the slave.
3. if the actor has no plans to execute, the `noPlan_event` is produced and the plan library is searched for the plan with default behavior of the actor.

```
process_event(Event):
    if peek($L_L$) != 0:                                        (1)
        ApplicablePlan = find_plan(+, Event, peek($L_L$))
        if ApplicablePlan != 0:
            new_strong_child(ApplicablePlan, peek($L_L$))
    else:                                                       (2)
        ApplicablePlan = find_plan(+, Event)
        if ApplicablePlan != 0:
            new_intention(ApplicablePlan)
end
```

Processing of an event depends on the current plan of the agent. If the actor already executes some plan, it searches for a new plan in the context of the current plan (1). Otherwise it just searches the actor's plan library (2).

```
find_plan(Trigger, Term):
    if peek($L_L$) != 0:
        return find_plan(Trigger, Term, peek($L_L$))
    else:
        searches the plan library for an applicable plan for Trigger/Term
end
```

```
find_plan(Trigger, Term, Node):
    while Node != 0:
        searches subplans of the Node.plan for an applicable plan for the
        Trigger/Term; if the plan is found, return it
        if Node.plan.hasAnnotation(
            PropagateEvents == false
        ):
            return 0
        Node = Node.parent
    searches the plan library for an applicable plan for Trigger/Term
end
```

When the event is processed in the context of some node, the subplans of the node's plan are first searched for the applicable plan. The algorithm then continues with node's parents. Every plan may contain an annotation PropagateEvents == false to prevent the propagation of the event higher. This may be used to create uninterrupted plans.

```
new_strong_child(NewPlan, Node):
```

The `NewPlan` is wrapped with the intention tree node and added as a strong child to the `Node`.

```
end
```

new_intention(NewPlan):

New intention tree is inserted into actor's F_I. The created tree has a `NewPlan` in the root.

```
end
```

```
execute(Plan):
     if Plan is template && Plan.executor != self:
          return
     if plan early success condition evaluates to true:
          succeed(Plan)
     if plan during condition evaluates to false:
          fail(Plan)
     if plan timed out:
          fail(Plan)
     execute_next_batch(Plan)
end
```

Every time a plan is scheduled for execution its early success condition, during condition and timeout is checked (if defined). Template plan is executed only by its executor, if the agent is the slave of the current plan, it does nothing.

```
execute_next_batch(Plan):
     Batch = plan.next_batch()
     while Batch.hasNext():                                (1)
          if Plan != peek(L_L):                            (2)
               pushBack(Plan, Batch)
               return
          Expression = Batch.next()
          Batch.remove()
          evaluate(Expression)
     if Plan.noMoreBatches():
          succeed(Plan)
end
```

The batch is executed as long as 1) it contains next action (1), 2) the batch's plan is the same as the leaf plan with highest priority (2).

`evaluate(Expression)` may encounter these actions:

1. `belief base query / belief base change`

2. `method call / java expression / if-then-else / etc.`

StorySpeak defines various methods for performing story actions.

3. `parallel expressions`

Perform evaluation of expression batches in parallel. The execution of the actions will not wait for the parallel expression to finish. Nevertheless, the parallel expression returns a handle, which the user may synchronize on calling Java methods `waitAll()` or `waitOne()`.

4. `plan call`

   Searches the plan library for the plan to instantiate. If plan is not found, fails the plan. If plan is found, creates new plan node and connect it with current plan node with a strong link. Updates L_L of the executor.

5. `template plan call`

   As 5) but updates L_L of slaves as well.

6. `plan delegation`

   As 5) but connects the new node with weak link and updates L_L of the agent the plan is delegated to.

7. `template plan delegation`

   As 5) but connects the new node with weak link and updates L_L of the executor and slaves.

Execution of the batch of actions may have these results:

1. `no result`

   The batch has been evaluated and it was not the last one.

2. `the plan has succeeded`

   The plan may succeed due to these reasons:

   a) there are no more batches of actions to execute,
   b) method success() is called,
   c) early success condition of the plan evaluates to true.

3. `the plan has failed`

   Plan may fail due to the one of these reasons:

   a) story action fails,
   b) method fail() is called,
   c) plan timed out,
   d) during condition evaluates to false

```
succeed(Plan):
      if Plan.node is strong leaf:
            removes the Plan.node from the intention tree, removes Plan from $L_L$ of
            executor (and slaves)
      elseif Plan.node is weak leaf:
            removes Plan from $L_L$ of executor (and slaves), hibernates Plan.node
            (will be removed automatically when it becomes a strong leaf)
      else:
            can't reach here as only leaf plans may be executed
end
```

```
fail(Plan):
     remove all Plan.node children, update respective $L_L$    (1)
     if Plan is deletion plan:
          remove Plan.node, update $L_L$
          fail(Plan.node.parent)                               (2)
     else:
          ApplicablePlan =                                     (3)
               find_plan(-, Plan.term, Plan.node.parent)
          if ApplicablePlan != 0:
               Plan.node.switchPlan(ApplicablePlan)
          else:
               remove Plan.node, update $L_L$                  (4)
               fail(Plan.node.parent)
end
```

When the plan fails, it removes recursively all `Plan.node` children updating the lists of leaf plans (1).
If the plan is a deletion plan, it fails completely and propagates the failure to the plan's node parent (2).
If the plan is addition plan, it tries to locate a deletion plan and switch the plan inside the nodes. (3) If deletion plan is not found, it fails completely and propagates the failure to the plan's node parent (4).

This ends the explanation of the interpretation algorithm of StorySpeak.

9.12 Extending StorySpeak

StorySpeak is designed to be easily extended by the author–based methods and libraries. There are two ways how to do it.

The first way is to create custom methods that may be called directly from StorySpeak at any place where the expression is accepted (plan context, during/early success conditions, plan's body). The extension is done by subclassing the SPMethod class and definition of the `evaluate()` method. The method has access to the full context of the method execution and may even alter the stack of actions of the instantiated plan allowing the author to schedule and execute arbitrary code.

The second way is through the means of global variables and custom Java libraries. It is easy to create an object before the execution of the story and insert it into the framework as a global variable under a certain name. Such variable will be accessible at all places where the variable is allowed (plan context, during/early success conditions, plan's body). Additionally, StorySpeak is able to call objects' Java methods and access their public field.

10 Storytelling framework

Finally, we have assembled all pieces of the puzzle together. The storytelling framework should be utilized by the author as follows:

1. Use UT04 map editor to create a desired environment.
2. Define story places.
    - The UT04 represents the environment as a graph of navigation points. The author may assign a name to the list of navigation points. The framework will automatically provide facts `at(place_name)`.
3. Define story actions.
4. Define story relations and provide Hammurapi rules allowing the framework to infer them.
    - The framework will automatically take care of them, raising all events, modifying belief bases, etc.
5. Define roles and a plot using the StorySpeak language.
6. Create a simple XML file with the story configuration.

10.1 Additional infrastructure work

Except the implementation of StorySpeak interpret, there is a lot of infrastructure work that allows simple handling of the story facts and relations.

Firstly, the framework must provide automatic translation of Java beans into Prolog terms (Java collections / Prolog lists are not supported yet) and vice versa. Even more, these Java beans will be author–defined (story relations) therefore a general mechanism must have been provided.

Secondly, the framework must provide methods for inserting, updating and removing facts from the system. Every such operation triggers the update of appropriate belief bases, Hammurapi rules and produce event for StorySpeak. Additionally, all facts that are accessible by the actors must be reachable by the story manager as well.

Thirdly, the framework provides the information about the story fact and relation origin and its state. The state can be:

1. new – the fact or relation has just appeared
2. updated – the fact or relation has been updated
3. dropped – the fact or relation has been dropped

The fact/relation state information is crucial for the situations matching as the actor may need to perform differently in the case (for instance) that somebody approaches the actor ("near" fact is created) and when somebody leaves the vicinity of the actor. The author would have to simulate it with three different facts without this mechanism.

Fourthly, the implementation is multi–threaded to utilize modern CPUs with multiple cores. The implementation is much more difficult then it would be for the single thread application.

Fifthly, there are a lot of StorySpeak methods that allows the author to perform story actions in various ways (blocking, non–blocking, sequences, etc.).

Sixthly, the framework offers logging as a debugging tool. The framework is logging:

1. the reasoning of the actors – which events it process and plans it executes,
2. belief base queries and alterations,
3. execution of the story actions,
4. information about the performance of the whole framework (storing time that was needed to process facts, perform reasoning and execute actions).

Performance analysis

The performance analysis is offering this list of values (taken from the output of the utility program for performance analysis).

```
General legend:

Iterations#           ... number of iterations of reasoning counted
Format of values      ... min  <  average +/- variance  >  max

Story manager performance legend:

Total  (ms)           ... total time of one iteration of SM
Rules  (ms)           ... time of Hammurapi rules (SM perception)
Reason (ms)           ... time of StorySpeak reasoning iteration
Rules objs#           ... number of objects inside Hammurapi rules
                          (SM perception)
Believes  #           ... number of facts inside SM belief base

Actor performance legend:

Total  (ms)           ... total time of one iteration of actor logic
Batch  (ms)           ... time of -batch processing (messages from
                          GB2004)
Messages  #           ... number of messages processed during the
                          GB2004 batch
Facts     #           ... number of facts that has been changed
                          (inside rules/belief base)
                          during the batch
Rules  (ms)           ... time of Hammurapi rules (actor perception)
Reason (ms)           ... time of StorySpeak reasoning iteration
Action (ms)           ... time of story actions
Rules objs#           ... number of objects inside Hammurapi rules
                          (actor perception)
Believes  #           ... number of facts inside director belief
                          base
```

11 Evaluation

The storytelling framework is evaluated with four simple story scripts that are presenting its features. Each script is targeted at specific feature(s). I will provide their description according to the intended workflow as presented in chapter 9.

Every story script contains only one image snapshot that does not tell much about the actors' behaviors. More snapshots may be found in the Appendix B. There are also videos available on the enclosed CD that illustrates the behaviors much better.

All performance tests from this chapter were run on the notebook ASUS M50Vc, Intel Core2 Duo P7350 2GHz, 3GB RAM and Windows Vista 32–bit. The UT04 dedicated server were running on the same computer (without UT04 gui that eats a lot of system resources). Every story script was evaluated for two minutes.

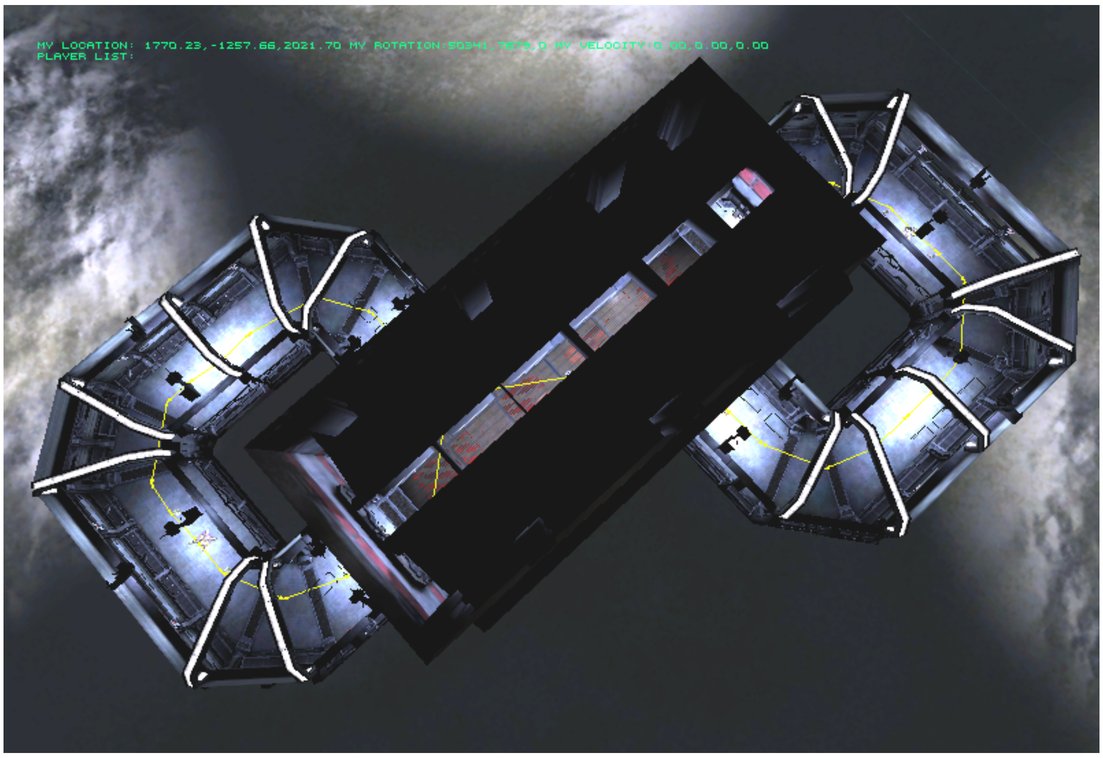
The legend for the performance analysis output is available in previous chapter.

11.1 Shared parts

Much of the code base is shared by all story scripts on the first three abstraction layers.

Virtual environment

All scripts are using one of the default map from the UT04 and that is DM–TrainingDay. The map is very basic thus suitable for the evaluation as actors may meet each other very often.


```
MY LOCATION: 1770.23,-1257.66,2021.70 MY ROTATION:50341,7679,0 MY VELOCITY:0.00,0.00,0.00
PLAYER LIST:
```

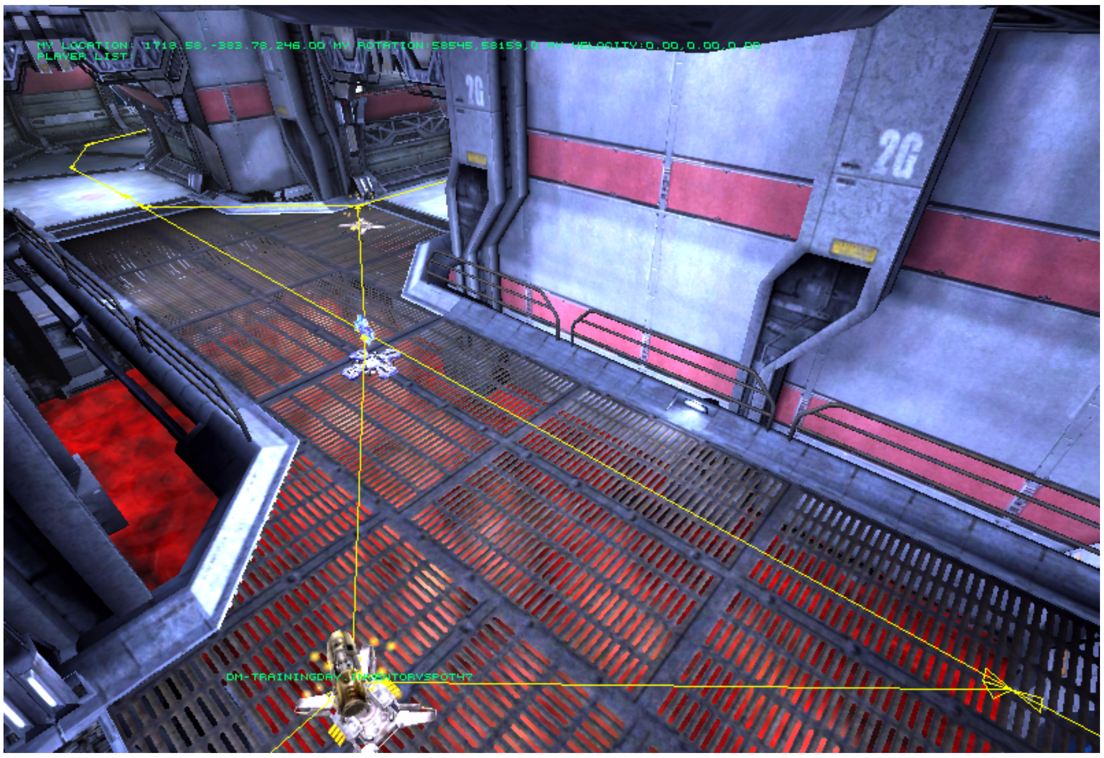

MY LOCATION: 1718.58,-383.78,246.00 MY ROTATION:58545,58159,0 MY VELOCITY:0.00,0.00,0.00
PLAYER LIST:
DM-TRAININGDAY.INVENTORYSPOT47

Figure 22 – Map DM–TrainingDay, yellow lines represents the default path the actors are walking along (GB04 feature).

Actor perception – base story facts

The UT04 allows actors to sense those story facts[24]:

1. `self(state, origin, location(x,y,z), rotation(roll, yaw, pitch), velocity(x,y,z))`
    - contains information about the actor itself, updated every 200ms
2. `see(state, origin, object, location(x,y,z), rotation(roll, yaw, pitch), velocity(x,y,z))`
    - information about the object or person that is in the actor's field of the view
    - objects may be:
        - `person(name)`
        - `item(type, ut04identifier)`
3. `hear(state, origin, from, to, text)`

Actor's perception – story relations

1. `at(state, origin, place)`
    - provides relation between actor's position and labeled place inside the story world, places are configurable

[24] Note that the facts are included only to make following story scripts more readable, I do not include, for instance, type information as they can be easily obtained from the Javadoc.

2. `near(state, origin, object, object)`
   - information that two objects are near to each other
   - objects may be:
     - `person(name)`
     - `item(type, ut04identifier)`

Story actions

1. `run around map`
   - moves around the provided location at random
   - implements dodging behavior
2. `turn to`
   - turns to other objects in the story world (person or item)
3. `say`
   - actor says aloud some message

11.2 Story script 1 – Simple greetings

The first story script is featuring two simple plans. First one is triggered whenever the actor has no plan and the second one when the actor become near to another player. Note that this player does not need to be another actor but human player as well.

Demonstrated features

1. actors may use story relations as a trigger for the behavior
2. actors may be given a default behavior when no situation is matching
3. Prolog unification of events with plan's head
4. various expression features of StorySpeak

Actors' plans
All actors in this story script have these two plans:

```
+!near("New", self, self.person, person(Name)) [Priority = 10]          (1)
      <-
            do( self.actions.say( text("greeting", Name) ) );           (2)
#

+!noPlan() [Priority = 1]                                               (3)
      <-
            perform( self.actions.runAround( place("map") ) );          (4)
#
```

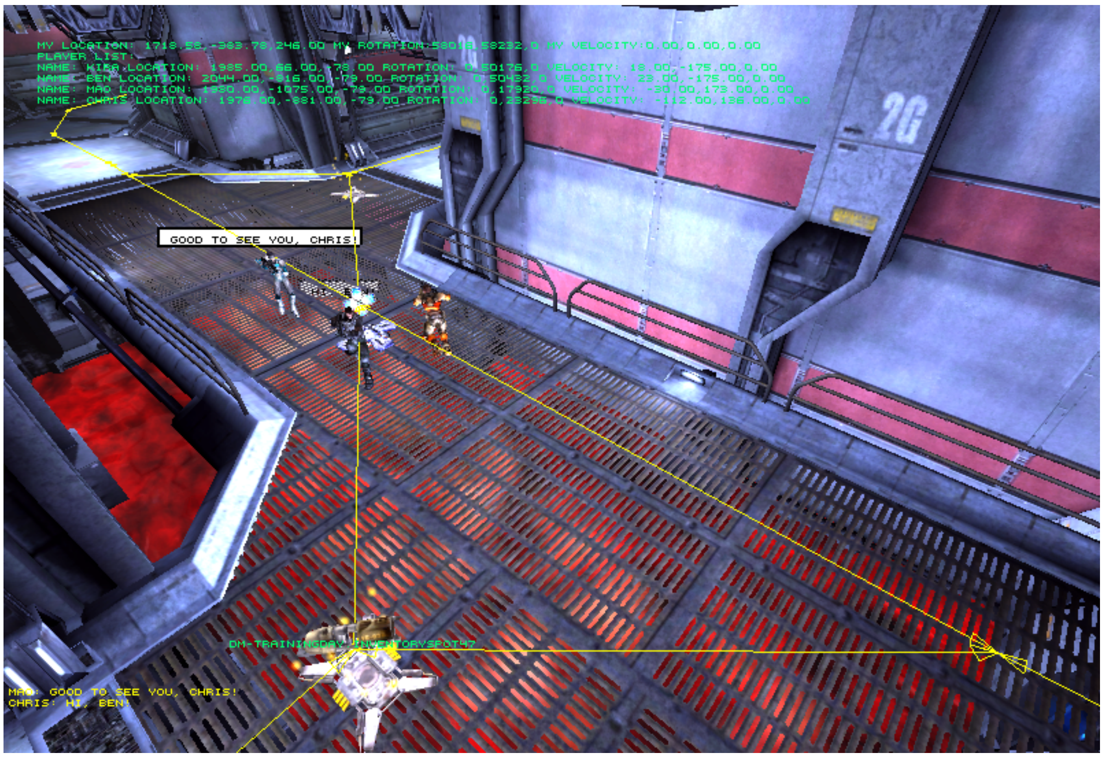
There is one situation/behavior pair specified at (1) and default behavior for the actor at (3). Translated to English, the (1) is saying: whenever a story relation `near` appears as a "`new`" fact (1st argument) in the belief base, it is me who sense it (2nd argument), it is me who is near something/somebody (3rd argument) and the other object is another `person` with some `Name`, then greet the person (2).

The default behavior is just performing the running around the map (4).

Notice the `Priority` annotations of the plans that are saying that the behavior triggered by the `near` fact has the bigger priority then the default behavior.

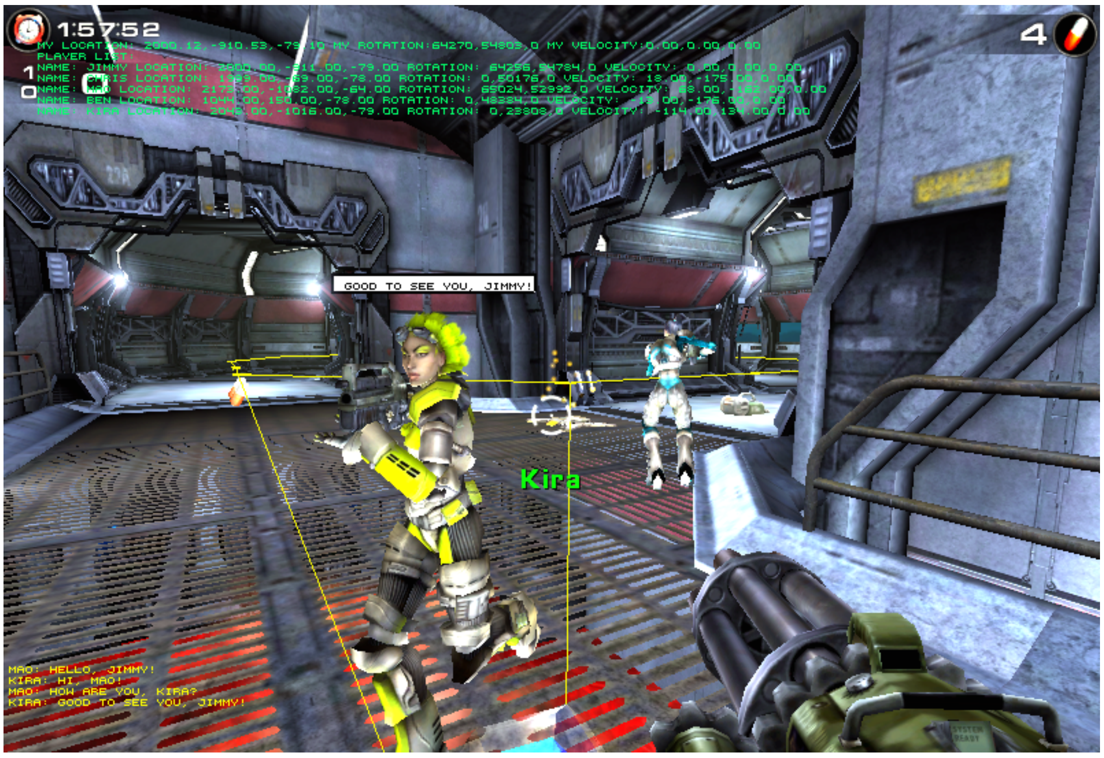
(1) is demonstrating feature 1 and 3, (3) is demonstrating feature 2, (2) and (4) is demonstrating the feature 4.

Snapshots



```
MY LOCATION: 1718.58,-383.78,246.00 MY ROTATION:58018,58232,0 MY VELOCITY:0.00,0.00,0.00
PLAYER LIST:
NAME: KIRA LOCATION: 1985.00,66.00,-78.00 ROTATION: 0,50176,0 VELOCITY: 18.00,-175.00,0.00
NAME: BEN LOCATION: 2044.00,-816.00,-79.00 ROTATION: 0,50432,0 VELOCITY: 23.00,-175.00,0.00
NAME: MAO LOCATION: 1930.00,-1075.00,-79.00 ROTATION: 0,17920,0 VELOCITY: -30.00,173.00,0.00
NAME: CHRIS LOCATION: 1976.00,-881.00,-79.00 ROTATION: 0,23296,0 VELOCITY: -112.00,136.00,0.00
```

```
MAO: GOOD TO SEE YOU, CHRIS!
CHRIS: HI, BEN!
```



```
MY LOCATION: 2000.12,-910.53,-79.10 MY ROTATION:64270,54803,0 MY VELOCITY:0.00,0.00,0.00
PLAYER LIST:
NAME: JIMMY LOCATION: 2000.00,-911.00,-79.00 ROTATION: 64256,54784,0 VELOCITY: 0.00,0.00,0.00
NAME: CHRIS LOCATION: 1999.00,-69.00,-78.00 ROTATION: 0,50176,0 VELOCITY: 18.00,-175.00,0.00
NAME: MAO LOCATION: 2173.00,-1082.00,-64.00 ROTATION: 65024,52992,0 VELOCITY: 68.00,-182.00,0.00
NAME: BEN LOCATION: 1044.00,150.00,-78.00 ROTATION: 0,48384,0 VELOCITY: -13.00,-176.00,0.00
NAME: KIRA LOCATION: 2043.00,-1016.00,-79.00 ROTATION: 0,23808,0 VELOCITY: -114.00,134.00,0.00
```

```
MAO: HELLO, JIMMY!
KIRA: HI, MAO!
MAO: HOW ARE YOU, KIRA?
KIRA: GOOD TO SEE YOU, JIMMY!
```

Figure 23 – Actors are meeting in the center of the map in the first picture. The second picture shows how actor may react to the player Jimmy.

Performance analysis
The performance analysis contains values from one actor only because all actors are running according to the same plans.

```
Time of simulation: 119,697 secs

Actor 'Mao' performance ...

Iterations#:     598
Total  (ms):   3,785 <   16,300 +/-   13,357 >   75,723
Batch  (ms):   2,753 <   13,190 +/-   10,595 >   72,539
Messages  #:      13 <   21,371 +/-    3,724 >       36
Facts     #:       1 <    2,262 +/-    1,249 >       12
Rules  (ms):   0,007 <    0,368 +/-    1,477 >   17,456
Reason (ms):   0,398 <    2,056 +/-    5,088 >   22,531
Action (ms):   0,000 <    0,687 +/-    1,067 >   17,130
Rules objs#:   4,000 <   10,162 +/-    3,746 >   23,000
Believes  #:   3,000 <    9,190 +/-    3,721 >   22,000

Longest batch:
Perf: total 75,723 ms, batch 70,215 ms (m20, f1), rules 0,045 ms,
reason 5,138 ms, action 0,326 ms, ruleobjs 6, believes 5
```

The longest part of the whole iteration takes the processing of the GB04 batch of messages. This number is a bit strange as it sometimes takes even 170ms to finish and depends more on the nature of GB04 then storytelling framework. The GB04 sometimes pauses the production of the messages for a certain amount of time, which results in such values.

11.2 Story script 2 – Greetings with replies

The second story script is extending the plan for greeting. The actors will now wait for the reply. If the other actors replies to the greeting (saying anything), they will say good bye to each other and continue their walking.

Demonstrated features

1. extensibility by additional story relations
2. plan context
3. belief base querying
4. belief base alteration
5. synchronization of the actors through the means of `waitfor` method
6. ternary operator implementing if–then–else

New story relation
This story script is introducing another story relation - “heard”. This relation is tracking the last heard sentence of other actors as well as the time when the sentence was heard.

Actors' plans

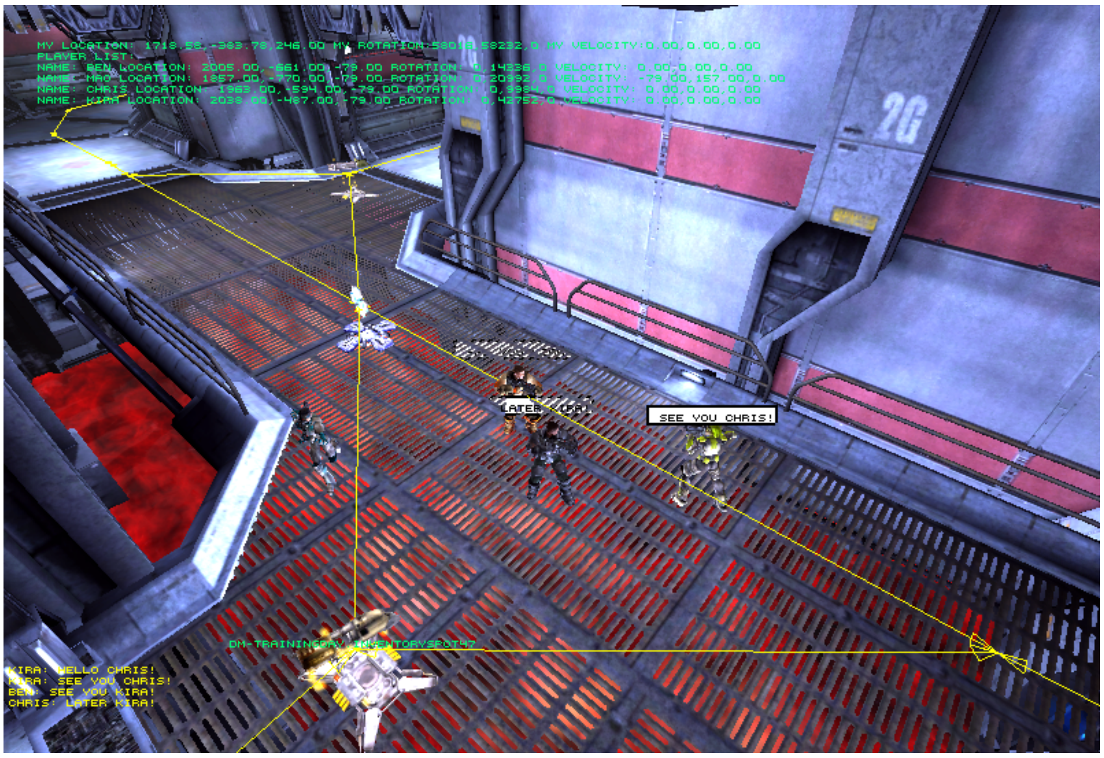
All actors in this story script has these two plans:

```
+!near("New", self, self.person, person(Name)) [Priority = 10]
      :
            ?greeted(Name, Time) ? Time < time()-50 : true        (1)
      <-
            self.actions.clear(),
            Time ? -greeted(Name, Time),                          (2)

            sequence(                                             (3)
                  self.actions.face( actor(Name) ),
                  self.actions.say( text("greeting", Name) )
            ),

            +greeted(Name, time()),                               (4)

            waitfor(                                              (5)
                     ?heard(_, self, Name, To, _, HeardTime)      (6)
                  && (To == self.name || To == "everyone")
                  && time()-3 < HeardTime,
                  3
            ) ?
                  do( self.actions.say( text("bye", Name) ) )
            :     do( self.actions.say( text("ignore", Name) ) );
#

+!noPlan() [Priority = 1]
      <-
            perform( self.actions.runAround( place("map") ) );
#
```

The plan for meeting other person has been extended of a context. The context (1) is querying the belief base – asking, whether the actor already greeted the person with `Name`. If not – the context is valid, if so – the time of the greeting is checked and must not be older then 50 (measured in seconds) for the plan to be instantiated. There are also a few lines in the plan's body that deserves attention. (2) is querying whether the variable `Time` has been bound and if so removes the fact is removed from the belief base. (3) is performing StorySpeak method sequence that waits the end of the list of actions. (4) adds the information about greeting the person and (5), (6) wait for 3 seconds (the 2nd argument of the `waitfor` method) whether the other person replies (the reply should not be older 3 seconds), note that method `waitfor` returns true if the condition we were waiting for were satisfied and false otherwise.

(1) is demonstrating feature 2, (2) and (4) are demonstrating features 3 and 4, (5) is demonstrating feature 5, (2) and (6) are demonstrating feature 6. (6) is also demonstrating feature 1.

Snapshot



```
MY LOCATION: 1718.58,-383.78,246.00 MY ROTATION:58018,58232,0 MY VELOCITY:0.00,0.00,0.00
PLAYER LIST:
NAME: BEN LOCATION: 2005.00,-661.00,-79.00 ROTATION: 0,14336,0 VELOCITY: 0.00,0.00,0.00
NAME: MAO LOCATION: 1857.00,-770.00,-79.00 ROTATION: 0,20992,0 VELOCITY: -79.00,157.00,0.00
NAME: CHRIS LOCATION: 1963.00,-594.00,-79.00 ROTATION: 0,9984,0 VELOCITY: 0.00,0.00,0.00
NAME: KIRA LOCATION: 2038.00,-487.00,-79.00 ROTATION: 0,42752,0 VELOCITY: 0.00,0.00,0.00
```

```
KIRA: HELLO CHRIS!
KIRA: SEE YOU CHRIS!
BEN: SEE YOU KIRA!
CHRIS: LATER KIRA!
```

Figure 24 – Actors are greeting and saying good bay to each other.

Performance analysis

```
Time of simulation: 119,925 secs

Actor 'Ben' performance ...

Iterations#:     599
Total   (ms):   2,803 <   16,047 +/-   12,467 >  102,070
Batch   (ms):   2,070 <   12,966 +/-   10,914 >   86,290
Messages   #:      13 <   22,687 +/-    3,944 >       32
Facts      #:       1 <    2,493 +/-    1,482 >       12
Rules   (ms):   0,007 <    0,263 +/-    0,850 >   13,846
Reason  (ms):   0,364 <    2,153 +/-    4,055 >   24,005
Action  (ms):   0,000 <    0,666 +/-    0,829 >    9,583
Rules objs#:    7,000 <   25,050 +/-    7,506 >   44,000
Believes   #:   6,000 <   11,947 +/-    4,016 >   32,000

Longest batch:
Perf: total 102,070 ms, batch 86,290 ms (m31, f6), rules 0,905 ms,
reason 14,482 ms, action 0,392 ms, ruleobjs 36, believes 14
```

Comparing numbers with previous results we may see that number of rules objects and believes increased. That is a result of the new story relation that is present inside belief bases as well as Hammurapi rules. Time of StorySpeak reasoning does not changed much with the presence of the context of the greeting plan.

11.3 Story script 3 – Story manager orders actors to party!

The third script introduces a story manager that contains a template plan that orders the actors to party when two of them meet somewhere.

Demonstrated features

1. story manager
2. plan start execution block
3. template plans
4. calling plan with a specific priority

Story manager plans

```
+!near("New", _, person(Name1), person(Name2))                      (1)
:
            !actor(Name1).locked                                     (2)
      &&    !actor(Name2).locked
      &&    (
            ?party(Name1, Name2, Time1)                              (3)
            ?
                  Time1 < time()-80
                  :
                  (
                        ?party(Name2, Name1, Time2)
                        ?
                              Time2 < time()-80
                        :     true
                  )
            )
<<
      actor(Name1).lock(), actor(Name2).lock()                       (4)
<-
      ?party(Name1, Name2, Time3) ? -party(Name1, Name2,
                                          Time3),
      ?party(Name2, Name1, Time4) ? -party(Name2, Name1,
                                          Time4),
      +party(Name1, Name2, time()),
      (2000) ! [actor(Name1), actor(Name2)] party();                 (5)
#

+![A, B]party()                                                      (6)
<-
      do( A.actions.say( text("party") ) ),                          (7)
      do( B.actions.say( text("party") ) ),
      do( A.actions.jump() ),
      A.unlock(), B.unlock();                                        (8)
#
```

Plans for the story manager have the same syntax as for the actor. The first plan (1) matches any `near` event that originates from any actor and has rather complex context. First we are checking whether the actors were already locked for the plan execution (2). If not we check whether they did not partied recently (3), notice the rather ugly double checking of the belief base. (4) contains expressions that are executed right after plan instantiation making the checking of the lock state and locking actions an atomic operation from StorySpeak perspective. (5) contains the template plan call with priority 2000.

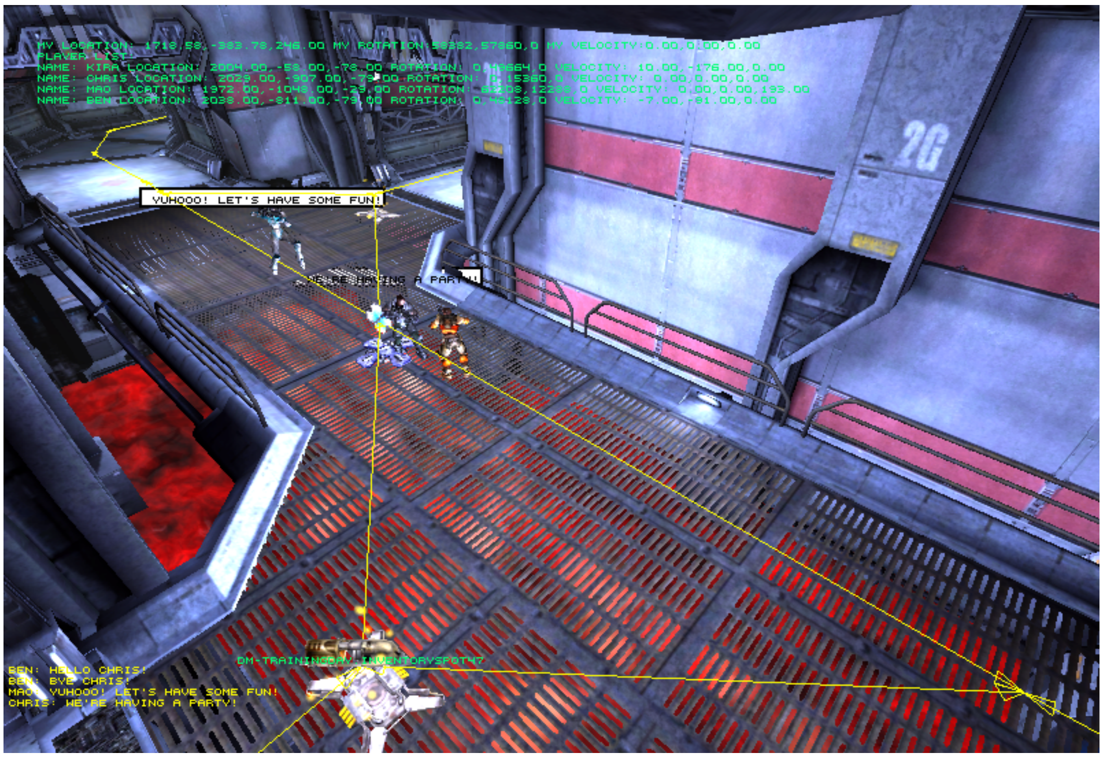
Next plan (6) is the template plan that has two slave actors – A and B. The plan (7) is ordering the actors (notice the use of actors' variables A and B) to say some cheerful sentence and one actor to jump. The plan ends with unlocking the actors (8) to allow them to party with somebody else.

The story manager has been introduced thus presenting feature 1. (4) demonstrates the feature 2. (5) is presenting feature 3 and (6) is presenting feature 4.

Actors' plans

Actors' plans are the same as in (ch 11.2).

Snapshot



```
MY LOCATION: 1718.58,-383.78,246.00 MY ROTATION:58382,57860,0 MY VELOCITY:0.00,0.00,0.00
PLAYER LIST:
NAME: KIRA LOCATION: 2004.00,-58.00,-78.00 ROTATION: 0,49664,0 VELOCITY: 10.00,-176.00,0.00
NAME: CHRIS LOCATION: 2029.00,-907.00,-79.00 ROTATION: 0,15360,0 VELOCITY: 0.00,0.00,0.00
NAME: MAO LOCATION: 1972.00,-1048.00,-29.00 ROTATION: 62208,12288,0 VELOCITY: 0.00,0.00,193.00
NAME: BEN LOCATION: 2038.00,-811.00,-79.00 ROTATION: 0,48128,0 VELOCITY: -7.00,-91.00,0.00
```

```
BEN: HELLO CHRIS!
BEN: BYE CHRIS!
MAO: YUHOOO! LET'S HAVE SOME FUN!
CHRIS: WE'RE HAVING A PARTY!
```

Figure 25 – Snapshot picturing the actor Mao in the middle of the jump saying a party message.

Performance analysis

```
Time of simulation: 120,336 secs

Story manager performance ...

Iterations#:    2347
Total   (ms):   0,020 <    1,808 +/-    5,968 >   93,411
Reason  (ms):   0,020 <    1,808 +/-    5,968 >   93,411
Active as.#:    0,000 <    0,020 +/-    0,142 >    1,000
Dormant a.#:        0 <          ...            >        1
Believes  #:   13,000 <   45,575 +/-   10,078 >   77,000
```

```
Longest batch:
Perf: total 144,411 ms,reason 144,411 ms, believes 76
```

The reasoning time of the story manager is very high sometimes – it is not because of StorySpeak but due to the synchronization. Whenever the story manager is executing the template plan it waits for the actor's avatar to finish his last action sometime.

11.4 Story script 4 – Ignorant Gregory

The last story script is not using the story manager but features two different kinds of roles. There are three actors Chris, Mao and Kira who are kind and reply to greetings. The last actor is Gregory that has been labeled as Ignorant because he never replies to greetings. That is because Gregory is too shy and runs away every time it gets near to somebody.

Demonstrated features

1. deletion (failing) plans
2. plan's during condition
3. subplans

Additional story actions

To allow Gregory to run we have to implement additional story actions.

1. `set walk`
   - sets the mode of moving to walking
2. `set run`
   - sets the mode of moving to running

Kind actors' plans

```
+!near("New", self, self.person, person(Name)) [Priority = 10]   (1)
:
      ?greeted(Name, Time) ? Time < time()-50 : true
!
      ?near("New", self, self.person, person(Name))              (2)
<-
      self.actions.clear(),
      Time ? -greeted(Name, Time),

      sequence(
            self.actions.face( actor(Name) ),
            self.actions.say( text("greeting", Name) )
      ),

      +greeted(Name, time()),

      waitfor(
               ?heard(_, self, Name, To, _, HeardTime)
            && (To == self.name || To == "everyone")
```

```
			&& time()-3 < HeardTime,
			3
		) ?
			do( self.actions.say( text("bye", Name) ) )
		:	do( self.actions.say( text("ignore", Name) ) );
#

-!near("New", self, self.person, person(Name)) [Priority = 10]   (3)
<-
	do( self.actions.say( text("rude", Name) ) );                (4)
	do( self.actions.jump() );
#

+!noPlan() [Priority = 1]
<-
	perform( self.actions.runAround( place("map") ) );
#
```

Kind actors' plans are similar to those from (ch. 11.2). The differences are in the head of the first plan and the deletion plan. The first plan (1) contains so–called during condition (2) that must be true during the execution of the plan. If the condition fails the plan fails. When that happens, the deletion plan (3) will be executed and the actor will say something rude about the person that has walked away and will also jump in fury.

(2) is presenting the feature 2, (3) is demonstrating feature 1.

Gregory's plans

```
+!near("New", self, self.person, person(_)) [Priority = 10]      (1)
<-
	self.actions.setRun();                                       (2)
#

+!near("Drop", self, self.person, person(_)) [Priority = 100]    (3)
:
	!(?near("New", self, self.person, person(_)))                (4)
<-
	waitfor( false, 2 ),                                         (5)
	self.actions.setWalk();                                      (6)
{
	+!near("New", self, self.person, person(_))                  (7)
	<-
		self.actions.setRun(),                                   (8)
		fail();                                                  (9)
	#
}
#

+!noPlan() [Priority = 1]
	<-
		perform( self.actions.runAround( place("map") ) );
#
```

Gregory is a shy guy therefore he pays attention to people that get too `near` to him by running away. Therefore there is the first plan (1) that will execute the (2) `setRun()` action which makes him run. When the `near` fact is “Drop”ped (disappears from the belief base) (3) and there is noone else who is `near` (4)

Gregory will keep running for two seconds (5) and then he will start walking again (6). But as was said in (ch. 9.11), whenever an event occurs it is matched in the context of the plan. Plans (except for `noPlan`) do not propagate events to the plan library. Therefore, we have to handle situation when the plan (3) is being executed and new `near` fact appears. Thus we will define a subplan (7) for plan (3) handling such appearance by start running (8) again and failing (9) the whole plan. This failure will be propagated to the higher plan (3) by StorySpeak tearing it down.

(7) is demonstrating feature 3.

Snapshots

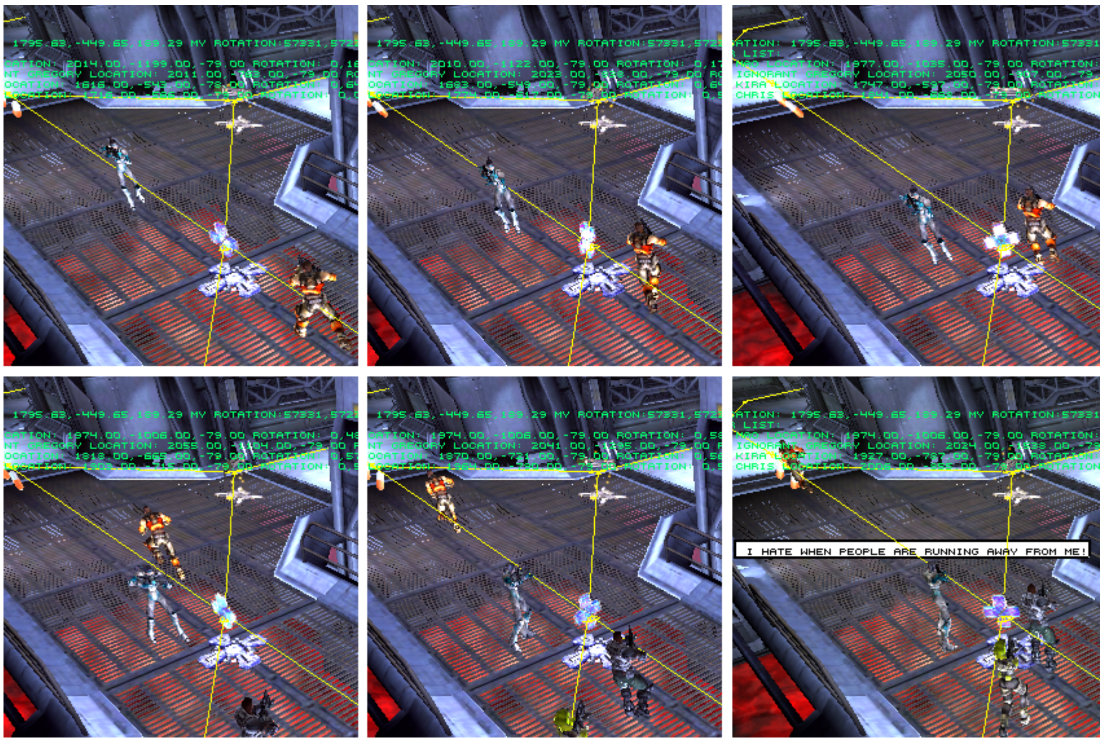


Figure 26 – A comics based on the presented story script featuring Furious Mao and Ignorant Gregory – perhaps Mao is a bit hot–headed due to the lava that is boiling under her feet?

Performance analysis

```
Time of simulation: 119,795 secs

Actor 'Mao' performance ...

Iterations#:     599
Total   (ms):   3,480 <    11,049 +/-     7,936 >    84,035
Batch   (ms):   2,073 <     8,367 +/-     6,704 >    81,457
Messages   #:      13 <    21,962 +/-     4,159 >        34
Facts      #:       1 <     2,533 +/-     1,414 >        12
Rules   (ms):   0,008 <     0,285 +/-     0,764 >    11,210
Reason  (ms):   0,382 <     1,861 +/-     2,968 >    33,467
Action  (ms):   0,000 <     0,537 +/-     0,669 >     5,216
Rules objs#:    6,000 <    26,372 +/-     8,852 >    47,000
Believes   #:   5,000 <    11,005 +/-     3,693 >    23,000
```

```
Longest batch:
Perf: total 84,035 ms, batch 81,457 ms (m27, f4), rules 0,170 ms,
reason 2,408 ms, action 0,000 ms, ruleobjs 36, believes 10


Actor 'Ignorant Gregory' performance ...

Iterations#:     599
Total  (ms):   2,893 <    8,591 +/-    4,278 >   21,983
Batch  (ms):   1,530 <    6,911 +/-    2,462 >   19,976
Messages  #:      13 <   21,448 +/-    4,115 >       31
Facts     #:       1 <    2,328 +/-    1,370 >        8
Rules  (ms):   0,007 <    0,250 +/-    0,563 >    4,094
Reason (ms):   0,276 <    1,050 +/-    2,287 >    5,903
Action (ms):   0,000 <    0,380 +/-    0,364 >    3,271
Rules objs#:   7,000 <   29,382 +/-   10,894 >   50,000
Believes  #:   5,000 <   11,023 +/-    3,648 >   25,000

Longest batch:
Perf: total 21,983 ms, batch 19,976 ms (m24, f4), rules 0,167 ms,
reason 1,551 ms, action 0,289 ms, ruleobjs 36, believes 18
```

There is no big difference between Mao and Gregory in execution times – except that Gregory was lucky and did not experience a GB04 lag.

12 Conclusion

This thesis has introduced the field of virtual interactive storytelling together with its two main problems: 1) narrative-interactive tension, 2) story definition. It was shown that the definition of the story is in fact definition of actors' behaviors. These behaviors are of two types: 1) interactive, 2) sequential. The author of the story has to be able to specify both of them. Therefore the actors' behaviors definition language must allow actors to switch between them. This was achieved by creating and implementing the StorySpeak language.

The story definition has been discussed in (ch. 6). It identified that the story definition has six layers of abstraction beginning in the virtual environment and ending at story execution. The layers are described in (ch. 6.6). These layers are:

1. Virtual environment
2. Sensing and acting in the environment
3. Actor's perception
4. Role definition
5. Plot definition
6. Story execution

The framework is built on top of Pogamut that utilizes Unreal Tournament 2004 virtual environment. Storytelling framework then allows the author to writer arbitrary number of story actions in Java using base commands that are recognized by the UT04. These story actions may contain reactive plans to express interactive behaviors.

The third layer has been implemented by using RETE algorithm implementation for inferring story relations. New rules and facts may be by the author as she see fit for the definition of the story world. Chosen RETE algorithm implementation is providing belief revisions of actors as well.

To facilitate fourth and fifth abstraction layer new language for actors' behaviors definition has been created that allows the author to write situation matching rules that triggers the behavior of the actor. The same language may be used to specify plans for the story manager that may use template plans to produce sequential behaviors for more actors and orders the actors to behave as the plot requires them to. The language allows the author to define situation-behavior pair.

Bringing everything together, the user may define simple XML files to define the whole story.

The storytelling framework has been evaluated with four short story scripts that have shown how behaviors may be specified using StorySpeak and how story manager may perform sequential behaviors with actors. During the evaluation the framework was extended by new story relation (heard), two new actions (set run and set walk) that has shown that the framework is easily extensible.

13 Future work

Currently, StorySpeak is not really suitable as an interactive behavior specification language. The interactive behavior must be implemented inside a story action. The future work should revisited the StorySpeak language and provide perhaps an additional grammar with refined language semantics to support reactive behaviors or utilize some reactive planner such as POSH [Byson01].

The utilization of tuProlog could also be improved. tuProlog is an open source Prolog implementation therefore it should not be that hard to extend it to support Java sets. Sets would be quite useful for simple specification of the symmetric facts (e.g. `near(Name1, Name2)` is the same as `near(Name2, Name1)`). This could also be solved by providing the author a way to define predicates that should be present in actors' belief bases.

Finally the next big goal would be to exploit use of planners and extend the definition of roles of constraints that the role wants to be held in the known state of the story world.

Literature

[Adobbati01]
Adobbati, R., Marshall, A. N., Scholer, A., and Tejada, S.: Gamebots: A 3d virtual world test-bed for multi-agent research. In: *Proceedings of the 2nd Int. Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, Montreal, Canada, 2001.

[Bae08]
Bae, B., Young, R.: A Use of Flashback and Foreshadowing for Surprise Arousal in Narrative Using a Plan-Based Approach. In: *Proceedings of First Joint International Conference on Interactive Digital Storytelling, ICIDS 2008*, Erfurt, Germany, 2006.

[Bordini06]
Bordini, R. H., and Hübner, J. F.: BDI agent programming in AgentSpeak using Jason. In: *Proceedings of the Sixth International Workshop on Computational Logic in Multi–Agent Systems (CLIMA VI)*, 143–164, 2006.

[BotPrize08]
http://botprize.org/2008.html [16. 4. 2008], computer bot contest, Perth, Australia, 2008.

[Bratman99]
Bratman, M.: Intention, Plans, and Practical Reason. CSLI Publications. ISBN 1-57586-192-5, 1999.

[Bryson01]
Bryson, J.J.: Inteligence by design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agent. PhD Thesis, MIT, Department of EECS, Cambridge, MA, 2001.

[Cavazza01]
Cavazza, M., Charles, F., J. Mead, S.: Characters in Search of an Author: AI–Based Virtual Storytelling. In: *Proceedings of the International Conference on Virtual Storytelling: Using Virtual Reality Technologies for Storytelling (ICVS01),* London, UK, 145–154, 2001.

[Clarke01]
Clarke, A., Mitchell, G.: Film and the development of Interactive Narrative. In: *Proceedings of the International Conference on Virtual Storytelling: Using Virtual Reality Technologies for Storytelling (ICVS01)*, Avignon, France, 2001.

[Dastani03]
Dastani, M., van Riemsdijk, B., Dignum, F., Meyer, J.J.: A Programming Language for Cognitive Agents: Goal Directed 3APL. In: *Proceedings of the First Workshop on Programming Multiagent Systems: Languages, frameworks, techniques, and tools (ProMAS03)*, 2003.

[Forgy82]
Forgy, C.: Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. In: *Artificial Intelligence*, 17-37, 1982

[Gazolla06]
Gazolla, G., Carrasco, R.: Implementation of Intelligent Agents to Unreal using Pogamut 2. Master thesis, Computer Science on the Federal University of Viçosa, MG, Brazil.

[Hubber99]
Huber, M. J.: International Conference on Autonomous Agents. In: *Proceedings of the third annual conference on Autonomous Agents*, Seattle, Washington, United States, 1999.

[Johnson07]
Johnson, W. L., Wang, N., Wu, S.: Experience with serious games for learning foreign languages and cultures. In: Proceedings of the SimTecT Conference, Australia, 2007 .

[Kadlec08]
Kadler, R.: Evolution of intelligent agent behaviour in computer games. Master thesis, Computer Science Department, Charles University of Prague, Czech Republic, 2008.

[Keller06]
Keller, J.: Úvod do sociologie, publisher SLON, ISBN 978-80-86429-39-7, 80-86429-39-3, EAN**:** 9788086429397, 2006.

[Kopp05]
Kopp, S., Gesellensetter, L., C. Krämer, N., Wachsmuth, I.: Conversational Agent as Museum Guide - Design and Evaluation of a Real-World Application. In: *The 5th International Working Conference on Intelligent Virtual Agents (IVA'05)*, 2005.

[Magerko05]
Magerko, B.: Story Representation and the Interactive Drama. In: *1st Annual Artificial Intelligence for Interactive Digital Entertainment Conference*, Marina del Rey, California, 2005.

[Rao06]
Rao, A. S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: Agents Breaking Away, Lectures Notes in Computer Science, Volume 1038/1996, 2006.

[Ruth06]
Hall, L., Woods, S., Aylett, R.: FearNot! Involving Children in the Design of a Virtual Learning Environment. In: *International Journal of Artificial Intelligence in Education*, Volume 16 , Issue 4, 327-351, 2006.

[Silva04]
De Silva, L., Padgham., L.: A Comparison of BDI Based Real–Time Reasoning and HTN Based Planning. In: *In 17th Australian Joint Conference on Artificial Intelligence,* 1167–1173, 2004.

[Turner92]
Turner, S.R.: Minstrel: A computer model of creativity and storytelling. *Technical Report UCLA-AI-92-04*, Computer Science Department, University of California, 1992.

[Riedel03]
Riedel, M. O., Young, R. M.: Character-Focused Narrative Generation for Execution
in Virtual Worlds. In: *Virtual Storytelling*, *Proceedings of ICVS 2003: International Conference on Virtual Storytelling*

[Wooldridge95]
Wooldridge, M., Jennings, N. R.: Intelligent Agents - Theories, Architectures and Languages. In: Volume 890 of Lecture Notes in Artificial Intelligence. Springer-Verlag, 1995.

Appendix A – Related work

I am aware only of two tools for creating virtual stories inside 3D environment: Machinima and Inscape.

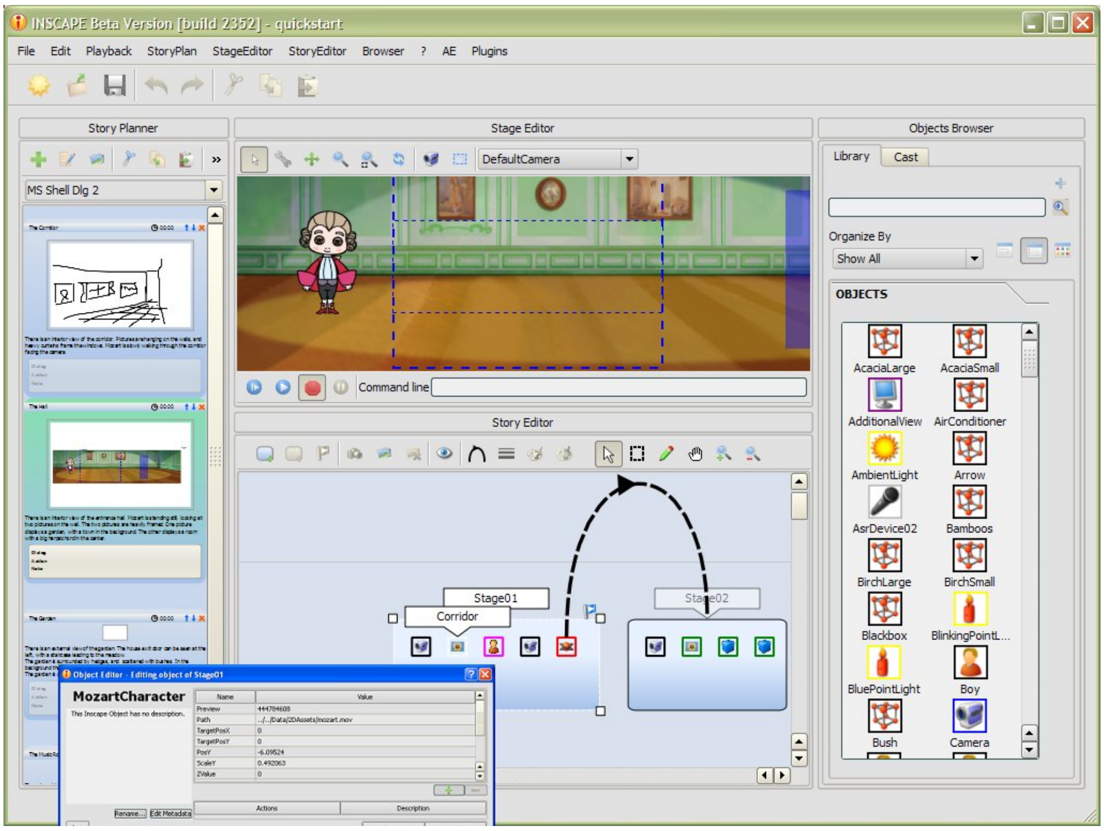
Machinima

Friedrich Kirschner's Machinima is a tool for making sequences of behaviors for avatars from Unreal Tournament 2004. The tool is allowing the user to specify sequences of actions that avatars should do. The definition is done inside UT04 environment where the user may directly specify the actions by interactively assign commands to the avatars.

Kirschner's Machinima is meant to produce in-game movies not interactive stories. The interactivity could be introduced only by coding the avatar directly in UnrealScript (native language of the Unreal Tournament 2004 game).

Inscape

The Inscape is the environment for the complete authoring of the virtual interactive story. It is an industrial platform therefore it offers visual authoring environment. The author may create the story starting with modeling the 3D virtual environment of the story and ending with the definition of the actors' behaviors. The Inscape is superior to the presented storytelling framework but it is an industrial platform that is closed-source and thus being in the different league.



This Inscape Object has no description.

Name	Value
Preview	444784608
Path	../../Data/2DAssets/mozart.mov
TargetPosX	0
TargetPosY	0
PosY	-6.09524
ScaleY	0.492063
ZValue	0

Figure A1 – The Inscape authoring environment

Appendix B – The StorySpeak language grammar

This chapter contains a specification of the StorySpeak grammar in EBNF[25] form (not entirely, some expansion are given in regular expressions[26], written in italic) that is widely used for that purpose. First I will provide the grammar without comments and then I will go through every derivation explaining it step–by–step.

Initial symbol of the StorySpeak language is `Plans`.

```
Plans = ( Plan '#' )*

Plan =
      ( '+' | '-' | '~' )
      '!'
      [ '[' Actors ']' ]
      Term
      [ '[' Annotations ']' ]
      [ ':' Context ]
      [ '!' DuringCondition ]
      [ '$' EarlySuccess ]
      [ '->' Variables ]
      [ '<<' StorySpeakExpressionSequence() ]
      '<-'
      PlanBody
      [ '{' Plans '}' ]

Actors =  Actor ( ',' Actor )*

Actor = ( SelfLiteral | Variable )

PlanName = Functor

Functor = ['a'-'z'] ( ['A'-'Z','a'-'z','0'-'9'] | '_' )*

Annotations = Annotation (',' Annotation )*

Annotation = Variable '=' Expression

Context = Expression

DuringCondition = Expression

EarlySuccess = Expression

Variables = Variable (',' Variable ) *

Variable = ['A'-'Z'] ( ['A'-'Z','a'-'z','0'-'9'] | '_' )*

PlanBody = StorySpeakExpressionSequences

StorySpeakExpressionSequences =

      StorySpeakExpressionSequence
      ( ';' StorySpeakExpressionSequence )*
      [ ';' ]
```

[25] As defined by ISO/IEC 14977, http://www.iso.ch/cate/d26153.html [13.11.2008]

[26] As used by JavaCC, https://javacc.dev.java.net/doc/javaccgrm.html [13.11.2008]

```
StorySpeakExpressions =
      StorySpeakExpression
      ( ';' StorySpeakExpression )*
      [ ';' ]

StorySpeakExpressionSequence =
      StorySpeakExpression ( ',' StorySpeakExpression )*

StorySpeakExpression =
      (
            ParallelExecution | PlanCall | BeliefChange
            AssignStorySpeakExpression
      )

ParallelExecution =
      [ Variable '=' ]
      '||'
      '(' StorySpeakExpressionSequences ')'

PlanCall =
      [ '(' Variables ')' '=' ]
      [ FixedPlanPriority ]
      [
            '[' Expression ']'
            |
            Variable
      ]
      ( '!!' | '!' )
      [ '[' ExpressionSequence ']'
      ]
      Term
      [ '/' 's' ]

FixedPlanPriority =

      '(' Expression ')'

BeliefChange =

      [
            ( '[' ExpressionSequence ']'
            |
            ( Variable )
      ]
      ( '+' | '-' )
      Term

AnonymousVariable =  '_'

ExpressionSequence = Expression ( ',' Expression )*

Expression = AssignExpression

AssignExpression =
      [
            Variable
            AssignOperator
      ]
      ConditionalExpression

AssignStorySpeakExpression =
      [
            Variable
            AssignOperator
      ]
      ConditionalStorySpeakExpression

AssignOperator =
      ( '=' | '*=' | '/=' | '%=' | '+=' | '-=' | '&=' | '^='
      | '|=' )

ConditionalStorySpeakExpression =
      ConditionalOrExpression
      [
            '?'
            (
                  '{' StorySpeakExpressions '}'
                  |
                  StorySpeakExpression
            )
            [
            ':'
            (
                  '{' StorySpeakExpressions '}'
                  |
                  StorySpeakExpression
            )
            ]
      ]

ConditionalExpression =
  ConditionalOrExpression
  [
      '?' ConditionalOrExpression
      ':' ConditionalOrExpression
  ]

ConditionalOrExpression =
  ConditionalAndExpression ( '||' ConditionalAndExpression )*

ConditionalAndExpression =
  ExclusiveOrExpression ( '&&' ExclusiveOrExpression )*

ExclusiveOrExpression =
  BeliefOrEqualityExpression ( '^' BeliefOrEqualityExpression )*

BeliefOrEqualityExpression():
      ( BeliefCheck | BeliefChange | EqualityExpression )

BeliefCheck =
      [ '<' ExpressionSequence '>'  | Variable ] '?' Term

PrologTerm =
      Term
Term = Functor [ '(' [ TermArguments ] ')' ]

TermArguments = TermArgument ( ',' TermArgument)*

TermArgument =
      ( MethodCall | Term | Expression | AnonymousVariable
        | Functor
      )
```

```
EqualityExpression =
      RelationalExpression
      ( ( '=='  | '!=' ) RelationalExpression )*

RelationalExpression =
      AdditiveExpression
      ( ( '<=' | '>=' | '<' | '>' ) AdditiveExpression )*

AdditiveExpression =
      MultiplicativeExpression
      (
            ( '+' | '-' ) MultiplicativeExpression
      )*

MultiplicativeExpression =
  UnaryExpression ( ( '*' | '/' | '%' UnaryExpression )*

UnaryExpression =
      (
            ( ( '+' | '-' ) UnaryExpression )
            |
            PreIncrementExpression
            |
            PreDecrementExpression
            |
            UnaryExpressionNotPlusMinus
      )

PreIncrementExpression = '++' PrimaryExpression

PreDecrementExpression = '--' PrimaryExpression

UnaryExpressionNotPlusMinus =
      (
            '!' UnaryExpression
      |
            PostfixExpression
      )

Expression = PrimaryExpression [ ( '++' | '--' ) ]

PrimaryExpression =
      ( MethodCall | Variable | Literal
      |
      '(' AssignExpression ')'
      )
      (
            '.' ( Functor | Variable )
            '(' [ ExpressionSequence ] ')'
            |
            '.' ( Functor | Variable )
      )*

MethodCall =
      Functor '(' [ ExpressionSequence ] ')'

Literal =
      (
            <INTEGER_LITERAL>
      |
            <DOUBLE_LITERAL>
      |
            <STRING_LITERAL>
      |
            BooleanLiteral
      |
            UnboundLiteral
      |
            NullLiteral
      |
            SelfLiteral
      )

BooleanLiteral =
  (
      'true'
      |
      'false'
  )

UnboundLiteral = 'unbound'

NullLiteral =  'null'

SelfLiteral = 'self'
```

Appendix C – Enclosed CD

The enclosed CD contains the sources of the StorySpeak along with a PDF version of this text. The directory structure of the CD is described in a *readme.txt* in the root directory. The sources provided on the CD can be used freely, without any license restrictions except for the Pogamut platform that has own license available at http://artemis.ms.mff.cuni.cz/pogamut.

The only request is to quote the author of this thesis when using any of his work in any way.