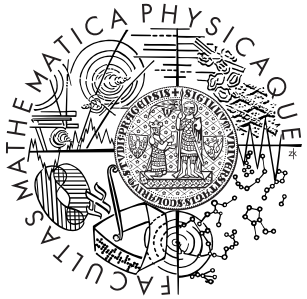
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Bc. Lucie Kučerová

Plánování osobní historie virtuálního agenta
Planning Personal History of a Virtual Agent

Department of Software and Computer Science Education

Supervisor: Mgr. Cyril Brom, Ph.D.

Study Program: Computer Science, Software Systems

2010

I would like to thank to my supervisor Cyril Brom for his unvaluable help during the preparation of the thesis, for his patient guidance and feedback. I would also like to thank to Rudolf Kadlec for his helpful suggestions.

I hereby claim that I have written this master thesis on my own, using exclusively cited sources. I permit the lending of the thesis.

Prague, August 6, 2010 Lucie Kučerová

Contents

Název práce: Plánování osobní historie virtuálního agenta
Autor: Lucie Kučerová
Katedra: Katedra software a výuky informatiky
Vedoucí bakalářské práce: Mgr. Cyril Brom, Ph.D.
E-mail vedoucího: brom@ksvi.mff.cuni.cz

Abstrakt: Episodická paměť je důležitou součástí „mysli" mnoha virtuálních agentů, protože agent s osobní historií bývá efektivnější a uvěřitelnější. Dosud se výzkum v oblasti modelování episodické paměti těchto agentů soustředil hlavně na vytváření obsahu paměti on-line, tj. během simulace agenta. V této práci se zabýváme příbuzným problémem, automatickým generováním obsahu paměti off-line. Pro designéra by bylo užitečné mít nástroj pro generování vzpomínek, které předcházejí startu simulace. Proto jsme vytvořili komplexní návrhovou metodu, která mu umožňuje specifikovat požadavky na historii agenta a použít plánování na generování historie v souladu s těmito požadavky. Zaměřujeme se na vysokoúrovňový jazyk používaný na popis požadavků a na část navržené metody, která se zabývá plánováním. V sadě experimentů jsme otestovali výkon několika plánovačů při řešení naší úlohy a představujeme zde výsledky, které jsme získali.

Klíčová slova: virtuální agenti, plánování, epizodická paměť, počítačové hry

Title: Planning Personal History of a Virtual Agent
Author: Lucie Kučerová
Department: Department of Software and Computer Science Education
Supervisor: Mgr. Cyril Brom, Ph.D.
Supervisor's e-mail address: brom@ksvi.mff.cuni.cz

Abstract: Episodic memory is an important component of "minds" of many long-living virtual agents, because equipping such an agent with his personal history increases his efficiency and believability. So far, research on episodic memory modeling in the context of these agents has focused mostly on producing the memory content on-line, that is, when the agent is being simulated. In this work, we address a complementary issue, automatic generation of the memory content off-line. We see a possible need of a tool for generating memories that anticipate the start of the simulation. Hence we created a complex design method enabling a designer to specify high-level requirements on an agent's history and use planning to automatically generate this history according to these requirements. We detail the structure of the high-level language used for the description of the requirements and the part of this method that concerns itself with the planning. In a set of experiments, we tested the performance of several planners on our task and we present here the results we gained.

Keywords: virtual agents, planning, episodic memory, computer games

Chapter 1

Introduction

One of the many effects of the growing computational power of personal computers in the last decades is that application designers are enabled to create and run simulations of complex virtual worlds. These worlds are inhabited by *intelligent virtual agents* (IVAs), intelligent agents [46] who are graphically embodied in the environment. Virtual worlds can be used for many purposes, for example therapeutic or educational. One of their important applications are computer games, notably *role-playing games* (RPGs). Contemporary RPGs feature vast virtual worlds inhabited by tens or hundreds virtual agents (e.g. [3, 18, 2]). The agents are also called *non-player characters* (NPCs), to differentiate them from characters controlled by a player.

As designing all the details of a virtual world can be a time-consuming, repetitive work, there are efforts to automatize it using procedural content generation. Currently, virtual world designers can use tools for automatic generation of textures [13], objects like trees or plants [37] and even large parts of a virtual world environment [21, 42]. However, there has been done significantly less research in the field of procedural generation of IVAs living in a virtual world. In this work, we are addressing one part of this task - automatic generation of the contents of episodic memory of virtual agents.

The term *episodic memory* was introduced for the first time by Endel Tulving in 1972 [44]. Today, psychologists divide human memory in three parts - episodic, semantic and procedural memory. *Procedural memory* stores mainly learned manual or physical activities. Riding a bike or swimming are typical examples of abilities included in procedural memory. *Semantic memory* is memory for knowledge about the world. Stores the facts that we learned, but that do not form a part of a particular context of our life. Examples of these facts are for instance the number of inhabitants of Prague or the year when Czechoslovakia was founded. *Episodic memory* stores our personal experiences and memories. Using our episodic memory, we can answer questions like "What did you have for lunch yesterday?", "When did you visit Madrid?" or "With who did you spent your last vacations?".

Recently, there has been increasing interest in modeling episodic memory for intelligent virtual agents. Some of that work focuses on on-line storage and later recall, including explaining by the agent (e.g. [5, 11, 38]). They argue that it may be vital to equip NPCs with episodic memory abilities [7, 12]. Among other aspects, equipping virtual agents with episodic memory can increase their believability [4]. An agent is considered believable if he allows the audience to suspend their disbelief

and if he provides a convincing portrayal of the personality they expect or come to expect [31].[1]

Other work in this field focuses on learning from past experience (e.g. [26, 34]), which can improve the performance of a virtual agent dramatically. For instance, an agent with the hunger drive can satisfy his eating need more quickly when he remembers where he saw food recently. We will present some of these works more in detail in Chapter 3.

To our knowledge, no one has addressed yet other issue concerning episodic memory modeling – the automatic off-line generation of the memory content in cases when it is impossible or inconvenient to create it manually. For instance, if an RPG designer wants the NPCs to have memories for events that anticipate the start of the game, she has to write them manually. While this approach is befitting in the case of main characters, it can get inconvenient in the moment when the designer has to model many NPCs of low importance (background characters throughout). Another possibility for the designer is to have at her disposition a method to generate these supplementary NPCs automatically according to her wishes, which is precisely the aim of our research now.

This issue of memory contents generation is scalable in at least two ways, the length of content of episodic memory and the number of agents whose histories are being generated. For example, we can generate an agent's memories for the entire lifetime, but we can also omit childhood or old age. As there are applications where it is not necessary to include these two periods, a work missing them out still remains usable. It is also different whether we generate memories of one agent, several agents that never interacted with each other during the time period in question, or hundreds of interconnected agents, i.e. agents who have relationships among them. Generating such agents separately could cause inconsistencies in their memories.

Each of these concepts brings different complexity. As this work is, to our knowledge, the first one addressing this problem, we intentionally focus here on the less complex variants. We create memories of an agent or a few interconnected agents spanning over several years of their adult life.

A typical agent with the content of episodic memory generated by our method would be a background character, e.g. a soldier, an ordinary mage, a villager or a shopkeeper in a fantasy RPG. Such RPG is just one of the possible applications, however, we will center mainly on this use in the text, for explanatory purposes. The agent would be able to tell the player a brief summary of his life or respond him to basic questions about his life. For instance, a soldier would be able to say in which battles he took part, when and where he got married or when he was ascended to lieutenant. One simplified history of a soldier which will be used as an example throughout the text is illustrated in Fig. 1.1.

We started to observe this agent when he was eighteen. About three months after that, he got money for studies and started to study military academy to become a soldier. This lasted almost two years. During his studies, he started to date Susan and they got married, approximately three months after finishing the military academy. And about a half year later, he took part in the battle for Suncity.

However, we cannot expect a game designer to describe her requirements on the

[1] This work contains verbatim or only slightly modified citations from the article [29], of which the author of this work is the main author. These citations are marked by a bar on the left side of the text.

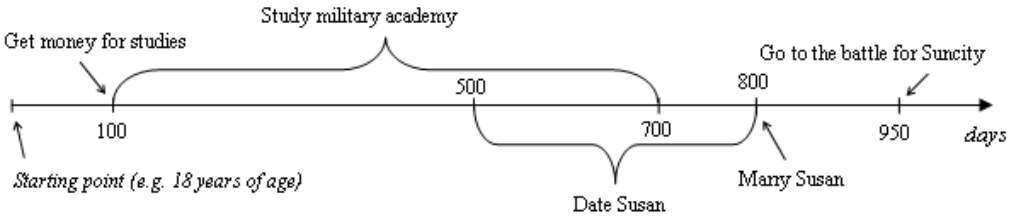


Figure 1.1: Schema of content of episodic memory. A real schema would be more complex; this one is simplified for intelligibility.

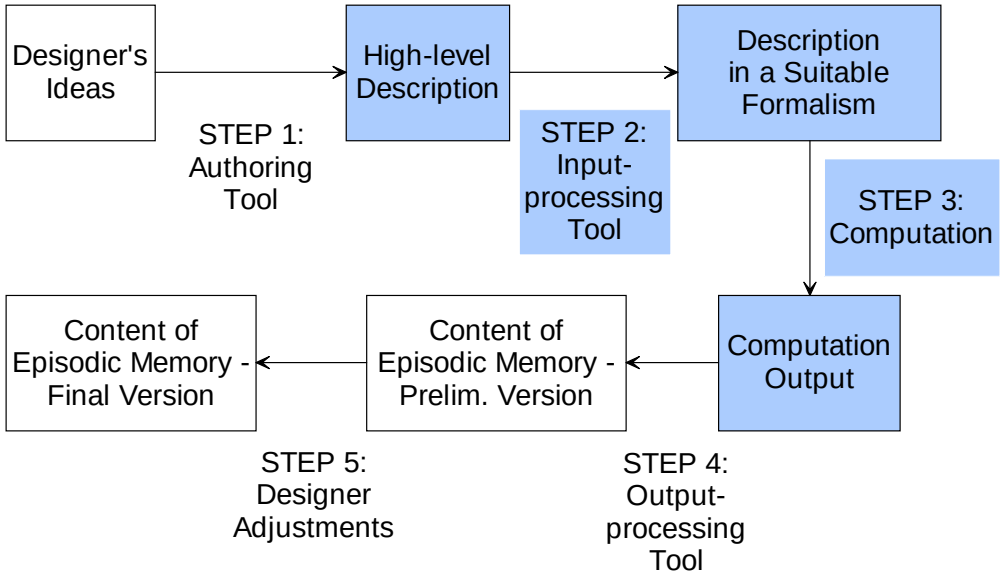


Figure 1.2: Method Workflow. The parts addressed in this work are marked with blue color.

history of an agent in a low-level formalism of the approach used to generate it. At the same time, the designer may want to fine-tune the generated history manually. Thus, we foresee a necessity of a complex design method enabling the designer to work with a sort of user-friendly software tool encapsulating the very process of generating an agent's memories.

Our proposal of a method fulfilling this is captured schematically in Fig. 1.2. We have to enable the designer to write down her intentions in a high-level, user-friendly language, using appropriate authoring tool. For this reason, we introduce Step 1 in our method workflow. This high-level definitions and requirements then have to be converted by an input-processing tool to a valid input for the program which will carry on the proper generation, which occurs in Step 2. This program afterwards generates the content of episodic memory of the agent described in its formalism (Step 3). Next, this output has to be converted back to the form suitable for the designer. This is done by an output-processing tool (Step 4). The designer then may want to make some manual changes to the created content of episodic memory. So we have supplied the workflow with Step 5.

The goals of this work are, aside from proposing the above mentioned design method, to present

1. a suitable high-level language for describing a designer's requirements (Step 2),
2. a proof-of-concept implementation of the generating part (Step 3) and
3. results of several case studies.

The rest of this work proceeds as follows: In the next chapter, we will reason about the method selected to solve our task. Chapter 3 presents related works. Chapter 4 details the requirements a designer may have on the contents of episodic memory to be generated. In Chapter 5, we will discuss the choice of a suitable tool to perform the generation and describe the main part of this work, the structure of the high-level designer language we developed and its transformation to the formalism of the selected tool. Chapter 6 presents the results we have obtained so far and a discussion of them. In Chapter 7 we will mention possible future works and then we will end by a short conclusion in Chapter 8.

Chapter 2

Method Selection

As the first step of our work, we had to choose a method which could be suitable for achieving our goal. We were considering simulation, constraint solving and planning. As the title of this thesis suggests, we decided to try the last one. However, it would be useful to summarize our reasons to do that.

2.1 Simulation

Simulation is a standard way to have the content of a virtual agent's episodic memory generated, because it is natural to create the memories of an agent during his "life". On the other hand, in our case this approach has several disadvantages.

One of the problems is the design of the simulation itself. It would be very difficult to describe a complex virtual world and the agents enough to produce realistic and varied memories for all the agents. But the main issue is that we do not want the agents just to have plausible history; we also want this history to fulfill a designer's requirements. We would have to assure for example that an agent would execute a given action at a certain moment in the simulation etc. As it seemed very unclear to us how to achieve this, we searched for another possibility.

2.2 Constraint Satisfaction

Other approach to this task is to use constraint satisfaction [43]. In contrast to simulation, constraints seem to be a good way to encode a designer's requirements. This method thus does not suffer from the disadvantages of the previous one. Unfortunately, designing a method to convert a complex virtual world with all its possibilities and flowing time to a constraint satisfaction problem is not very straightforward. So we decided to look for another possibility in the first place and to let CSP remain to be the second choice in the case that we did not find a more hopeful approach.

2.3 Planning

The idea to use planning in our work rose from the properties of the episodic memory itself. According to some, one of the main functions of episodic memory is social (e.g. [45]). In a nutshell, people tend to exchange their personal stories for various social reasons. As Hirst and Manier put it in [24]: "We cannot divorce the act

of remembering from the act of communicating. ... Recollections arise... from a desire to communicate with others about the personal past." The conceptualization of (some) recollections as personal narratives brings us to the idea of generating these recollections similarly to how narratives are generated in the field of virtual storytelling.

There are more approaches to generating narratives, but planning is arguably the most promising. In our domain, using planning would mean generating the content of episodic memory, i.e. the underlying representation enabling the agent to tell stories about himself, based on a planning domain and planning problem. This content would be generated based on the agent's initial state, his possible memories and a designer's requirements constraining the to-be-generated agent's memory. The core of a planning domain is formed by predicates and operators. Predicates can serve to describe possible agent's states; operators have preconditions and effects, which make them a suitable representation of actions in the virtual world, thereby of the agent's possible memories. A planning problem consists of description of the initial state and the goal state. These two can serve to represent the designer's constraints on the to-be-generated memory.

Weighing all the above mentioned aspects, we consider planning the most promising way to solve our problem, so we decided to use this approach.

Chapter 3

Related Works

Generally, the works related to our research are of two types. One of them is formed by works tackling episodic memory for virtual agents. The second one rises from the fact mentioned in the previous chapter - from one point of view, recollections are similar to narratives. So we also include some notable works from the field of automatic generation of narratives in the second part of this chapter.

3.1 Episodic Memory

As mentioned in the first chapter, recently there has been interest in providing virtual agents with episodic memory to enhance their performance in various tasks or to increase their believability. We are going to present some of these works here now which argue that this approach indeed works.

Andrew Nuxoll's Implementation of Episodic Memory

In his dissertation, A. Nuxoll states these benefits of episodic memory for people which could be also useful for virtual agents [34]:

Sensing:

- Noticing Significant Input – detecting what is important about a given situation by its relative familiarity
- Detecting Repetition – realizing when you are repeating the same series of actions and altering your behavior as a result
- Virtual Sensing – retrieving past sensing of features outside current perception that is relevant to the current task

Reasoning:

- Action Modeling – predicting the immediate outcome of your actions
- Environment Modeling – using past experience to predict how the environment will change

- Recording Previous Successes/Failures – using past performance to guide future behavior
- Managing Long Term Goals – keeping track of a plan and what steps in that plan have been accomplished so far
- Sense of Identity – understanding one's own behavior in relation to other agents

Learning:

- Retroactive Learning – reviewing experiences and learning from them when sufficient time (or another resource) becomes available
- Reanalysis Given New Knowledge – relearning from experience upon receiving new knowledge
- Explaining Behavior – reviewing your past actions to others for mutual benefit
- "Boost" Other Learning Mechanisms – provide a database of knowledge that can be manipulated by other learning mechanisms

In his work, Nuxoll compares the performance of simple virtual agents with the performance of virtual agents equipped with episodic memory. The tests are undertaken in two environments, Eaters and TankSoar. In both of them, the agents with episodic memory outperform the simple agents.

Wan Ching Ho's Implementation of Episodic Memory

Ho, Dautenhahn and Nehaniv also realized tests on the performance of simple virtual agents and agents equipped with episodic memory [25]. They implemented a simple virtual world containing a desert, an oasis, a mountain, a river with a waterfall and a lake. Each of this environments has specific features for the agents, e.g. there grow apple trees in the oasis and so an agent can satisfy his hunger there.

The authors performed a suit of tests featuring simple reactive agents, agents equipped with short term episodic memory (STM), agents equipped with long term episodic memory (LTM) and also agents equipped with both of them. They examined the influence of the use of episodic memory to the average lifespan of the agents, therefore to the ability to survive in the virtual world. The agents with episodic memory clearly outperformed the purely reactive agents, the agents equipped with both STM and LTM being the most effective.

Episodic Memory in FearNot!

FearNot! [10] is a computer application developed to help to reduce bullying problems in schools. The user of the application interacts with virtual agents representing victims of bullying, presumably learning in this way how to cope with this kind of issues. The virtual agents are equipped with a module for emotions to impersonate believable bullying victims.

In the version 2.0 of the program, episodic memory module is added to the agents [11]. It enables the agents to report to the user the experience from the last session,

including the reasoning why the agent did or did not follow the advices given by the user. This arguably increases the efficiency of the application.

Exercise Counselor in the "Virtual Laboratory" System

Bickmore, Schulman and Yin implemented in their Virtual Laboratory framework an application meant to promote exercise among its users [1]. The application features a virtual agent representing an exercise counselor who should motivate the users to interact with her daily and fulfill the objectives of the exercise program.

They decided to equip the agent with a set of short pre-written stories – episodes from life, linked together in function of common concepts. The aim was to create a more human-like agent who can use pieces of her memories to enrich her communication with the user. In the experiment lasting about 30 days, one half of the users communicated with an agent speaking in first person (and presenting so the stories as episodes from her life), the second half communicated with an agent using third person (and presenting the stories as episodes from life of her friend).

The results showed that users interacting with the agent speaking in first person were more engaged with her and used the application more frequently than users from the second group. However, ratings of agent dishonesty were not significantly different between the groups. This displaced the worry of the authors that people could perceive as a deception when they are told stories from her life by a virtual agent, who is not a living person.

RPG Actor with a Full Episodic Memory

Brom, Pešková and Lukavský developed a *full episodic memory* system for virtual agents [5]. By "full", they mean that this model represents a generic episodic memory which can store all events relevant to the agent, not only events specified ad hoc by the programmer. The memory is optimized for effective storing and retrieval, is not domain specific and includes a simple algorithm for forgetting. They performed various experiments with their memory model which show that equipping an agent with an implementation of this model makes him more efficient.

The aim of their ongoing research is to implement episodic memory for virtual agents that would fulfill the assumptions which a user has about the human episodic memory. It means that the virtual agent should be believable in this aspect. However, this task is very complex and so far the only way to tackle it is to pick just one aspect of human episodic memory at a time. An example of this approach is an extending work focused on human-like memory for time of past events [6].

3.2 Generating Narratives

As described in the previous chapter, generating narratives is a problem similar to generating contents of episodic memory of a virtual agent. There has been done a lot of research in this field and we will present here several representative works. That will then enable us to summarize the differences between generating narratives and generating history.

Tale-spin

Tale-spin [33] is a program for generating narratives created by James R. Meehan. He successfully used this system to generate simple, but coherent Aesop-style fables. A typical Tale-spin output, after being rewritten to English, is for example this story called "Joe Bear and Jack Bear" [33]:

> Once upon a time, there were two bears named Jack and Joe, and a bee named Sam. Jack was very friendly with Sam but very competitive with Joe, who was a dishonest bear. One day, Jack was hungry. He knew that Sam Bee had some honey and that he might be able to persuade Sam to give him some. He walked from his cave, down the mountain trail, across the valley, over the bridge, to the oak tree where Sam Bee lived. He asked Sam for some honey. Sam gave him some. Then Joe Bear walked over to the oak tree and saw Jack Bear holding the honey. He thought that he might get the honey if Jack put it down, so he told him that he didn't think Jack could run very fast. Jack accepted this challenge and decided to run. He put down the honey and ran over the bridge and across the valley. Joe picked up the honey and went home.

Tale-spin uses planning to generate a correct sequence of actions to accomplish a given goal from a given initial state, i.e. to accomplish the goal providing every action has it's logical preconditions fulfilled before it starts. The main goal of the story is always to satisfy a physical need of the main character - e.g. hunger (like in the example above) or thirst. Meehan's central point of interest in this work was to develop some mechanisms which would assure that a generated story would be a "good story" from the point of view of a reader. He focuses on how to specify motivation, relationships and personalities of the characters and how to assure that the resulting story will respect all this input.

Mexica

Mexica [35] approaches the problem of generating narratives in a different way. It attempts to model the process which is supposedly used by a human author. From a given initial state and action, Mexica generates a story employing the cycle of engagement and reflection. In each step, it compares the actual context with the contexts stored in its internal database of stories. The actions which were made in similar contexts are candidates to the action which should be undertaken now. From these candidates, the system selects the most suitable action considering:

- general constraints (availability of the actions)
- novelty of the resulting part of the story in comparison to other stories in the database, as we do not want to produce (almost) the same story again
- compliance with the dramatic arc (the tension to the reader has to increase at first and then decline in the end)

The authors called the system Mexica because they were generating stories about the Mexicas, the old inhabitants of Mexico. A sample story [35]:

> Jaguar_knight was an inhabitant of the Great Tenochtitlan. Princess was an inhabitant of the Great Tenochtitlan. Jaguar_knight was walking when Ehecatl (god of the wind) blew and an old tree collapsed injuring badly Jaguar_knight. Princess went in search of some medical plants and cured Jaguar_knight. As a result Jaguar_knight was very grateful to Princess. Jaguar_knight rewarded Princess with some cacauatl (cacao beans) and quetzalli (quetzal) feathers.

Thespian

Thespian [41] is a multi-agent system which uses simulation for interactive storytelling [20]. The main difference between Thespian and Tale-spin or Mexica is that Thespian counts with participation of a user-directed agent who interacts with artificial virtual agents in real time. Thus the story has to be adapted on-line to fit to his actions.

In this framework, a designer specifies goals and possible actions of each virtual agent. Then she creates alternative linear scripts of the desired paths of the story. When an agent gets the turn, he chooses from his possible action the one which helps him to achieve one ore more of his goals and is also compatible with one of the story paths.

Generating Narrative Versus Generating History

After describing several systems for generating narrative and presenting a closer look on this field, we will now summarize the differences between generating narrative and generating history.

Generally, these tasks are closely related. In both of them, we want to generate an ordered set of actions which contains given characters and fulfills logical constraints. For example, a character cannot get divorced before getting married.

In generating narrative, this task is complicated by the fact that the generated set of actions should be a "story". It means it should have the features a good human-written story has, like a dramatic arc or believable motivation of the characters to perform an action.

From this point of view, our problem is simpler. The agent we want to be equipped with the generated memories is a background character who will interact with the player just in several brief conversations. For this purpose, we do not necessarily need him to be able to explain his motivations for all his actions (although it would be beneficial), he is just expected to be able to narrate his memories in a simple way. The "story" of the agent is partly defined by the properties of the virtual world and by the requirements from the designer and we do not need to make it a good story.

We also do not need to count with the user interaction, as seen in the task of interactive storytelling.

On the other hand, we have more demands on the properties of the generated set of actions. The game designer typically has several requirements the character should fulfill to fit in the virtual world and his overall story. For example, the designer may want the character to be rich or to take part in a particular battle. In virtual storytelling, setting some requirements on the story course is also used some-

times (e.g. [41, 39, 36]), because it generally leads to more complex stories. However, it is not absolutely necessary to include this possibility in generating narrative; by contrast, in our task it is essential.

A major difference between generating narrative and generating memories for agents living in a virtual world comes up when we start to think about time. In the former case, we are generating a story, which is basically a sequence of events. It is not necessary to reason about the duration of actions. But when we want to generate the contents of episodic memory, it is indispensable to do it, to have the character fit to the world. For instance, the character can take part in a particular battle only when it is going on. Introducing durative actions also brings parallelism. We need it among actions of one single agent as well as among actions of different agents. This all raises computational complexity of the task a lot.

Other aspect of our work is introducing randomness to gain the possibility to generate several similar, but a bit different, contents of episodic memory from just one input. The designer often needs to generate a lot of similar characters who are just slightly different. For example, if she wants to generate memories for the members of the palace guard, she may generally want them to be soldiers who have taken part in several battles. We need to allow her to generate the memories of all of them from one input, including a mechanism to introduce random (but fully coherent) actions in memories of each of them. In the task of generating narratives, generating several different stories from one input is also useful [30]. However, there is no need to generate tens or hundreds of stories from one setting, as in our task.

To summarize, a quality framework for generating narratives which would include the possibility to introduce requirements on the story, take into account time and enable certain level of randomness would be a tool strong enough not only to generate history of background characters, but also main NPCs. However, the actual state of the art in the field of generating narratives does not bring us hope that so powerful tool will be developed during the several next years. That is why we want to create a less complex tool suitable for generating history of background characters to help the designers until the research in generating narrative provides them with more efficient tools.

Chapter 4

Designer's Requirements

Generally, in Step 1 of the workflow presented in Chapter 1, the designer must create high-level description of the virtual world, i.e. define its topology, objects and possible actions which can be executed by an agent. At the same time, she has to specify her requirements on a concrete agent or a group of agents. Then, in Step 2, this high-level input has to be transformed to a planning domain and a planning problem.

We analyzed the types of conceptual requirements which can be demanded by a designer to specify the properties of the contents of an agent's episodic memory and the properties of the virtual world in general. The results of this analysis are presented in this chapter. We will again employ the example of a soldier in a fantasy RPG, as a fantasy RPG was the scenario we used to contemplate about the properties of the virtual world desirable for our task and the requirements a designer can have.

4.1 Requirements on the Virtual World

The first essential part of the description of the virtual world is its topology (cities, villages, important places...) and the objects that the world contains. The designer will need a way to specify all of this. The other part is the specification of actions that are possible in the world, together with their preconditions and effects. Moreover, some real-like actions are naturally durative and some of them permit other actions to occur during their execution. For example, studying military academy takes some time and a future soldier can perfectly plausibly date a girl or win some money in roulette during his studies. So we have to permit durative actions in the world definition.

4.2 Requirements on an Agent

Conceptual requirements on the generation of history of an agent can be categorized into these groups:

1. General requirements on the agent's achievements or states.
2. Requirements on a concrete action.
3. Requirements on an action in a concrete time.

4. Randomness for the possibility to generate more agents from just one setting.

We will discuss these requirements in more detail now.

1. **General requirements.** The designer may wish to specify a general requirement on the end state of the agent, without assigning a particular way to achieve it. For example, she may want the soldier to be rich, letting completely to randomness whether he has gained the money by fighting, winning in a lottery or inheriting it.

2. **Requirements on a concrete action.** In some cases, the designer may want to specify a more concrete requirement: not only the end state, but also the means to achieve it. For instance, she may want the soldier to earn money by playing roulette, because it is important for the story.

3. **Requirements on an action in a concrete time.** Sometimes, it is important that an action occur in a concrete time. For example, the designer may want the soldier to take part in a particular battle. Since this can happen only in the moment when the battle takes place, there has to be a mechanism to accomplish this.

4. **Randomness.** The designer may need the agent to achieve a concrete end state (in a random way) or to perform a particular action. But she also may want to use randomness to generate some unspecified, random memories of the agent. In addition, for the purpose of saving her time, she may need to generate histories of several similar, but not identical, agents from the same high-level specification. Thus we need to introduce randomness in these two ways:

   (a) **Probability of the actions.** For example, it is a lot more probable to earn money for living by working than by finding them in a secret cave. So the designer can assign probability to each of these actions.

   (b) **Insignificant actions to "animate" a virtual agent.** The designer can mark some of the actions as "noise actions", which can be inserted to the history randomly (providing their preconditions are fulfilled). These could be actions like "go for a trip to the capital", "see a falling star" etc. They are absolutely not important for an agent's history, but can make him seem more vivid during the communication with a player.

Chapter 5

Planning

In this chapter, we will list the features that must be supported by a planner to enable its use in Step 3 of our method. Then we will discuss the selection of suitable planners. This will be followed by the description how can be a designer's requirements listed in the previous chapter translated to PDDL. Stemming from the observations made during the manual conversion of the requirements, we will continue by the key part of this work. We will define the structure of the high-level language suitable for specifying a designer's requirements and describe its conversion to PDDL.

5.1 Necessary Features of a Planner

The planning mechanism should cope with all of the requirements from the previous chapter. We also have to choose which formalism to use for its input. As PDDL seems to be actually the most used language for specifying planning problems, we decided to describe our domain and problems in this formalism [32].

Many requisitions on the used planner rise from the requirements described above. We will list them now in the form of PDDL requirements, divided in two groups – essential requirements, which are indispensable for our purposes, and technical requirements, which would be useful, but are not absolutely necessary.

Essential requirements.

- **:durative-actions** – This requisition stems directly from the requirements on the virtual world.
- **:fluents** – It is necessary to include several numeric variables, e.g. amount of money or a randomly generated number to introduce randomness in the generated problem/plan (see Req. 4).
- **:equality** – First, this is needed for randomness (Req. 4). Second, a designer may want an action to include preconditions comparing numeric variables with a predefined value. For example, she may want to specify that a soldier can be ascended to lieutenant only after taking part in a concrete number of battles.

Technical requirements.

- **:typing** – It is a lot more transparent to describe some preconditions of an action by specifying the type of parameters (e.g. action *get_ married(v1, v2)* does not make sense with locations as its parameters). However, if typing is not available, this can also be solved by inserting predicates of type *is_ person(v)*, although it makes the domain less human-readable.
- **:disjunctive-preconditions** – Many real-like actions may require disjunctive preconditions, nevertheless, these could be also formally written like several different actions.
- **:negative-preconditions** – There are many possible actions which need their preconditions to include negation of a predicate. However, this can be bypassed by inserting other predicates (for example adding predicate *not_ married ?person* to supplement a predicate *married ?person*).
- **:timed-initial-literals** – Timed initial literals are useful for Req. 3 from the previous section, but any PDDL domain containing timed initial literals can be transformed to an equal domain without them, as showed in [9].

5.2 Planner Selection

From the listed requirements it is obvious that we need a planner supporting all levels of PDDL2.2 [14], or at least PDDL2.1 [17], if we eliminate timed initial literals. In this, we depart from the work in generating narratives (e.g. [36, 40]), which usually does not demand so much equipped planner (although it tends to have other requirements, as mentioned in Chapter 3).

Implementing a satisfactory planner would be quite demanding, so we decided to use an already existing planner to prove feasibility of our concept. Sadly, we have not found many fully functional planners fulfilling all the essential requisites. At this moment, we are using SGPlan6 [28], Temporal Fast Downward (TFD) [16] and POPF [8] for our tests. We will now briefly present the approaches used by the planners to solve planning problems.

SGPlan6 employs *parallel decomposition* to partition a state space into subproblems. This leads to a partitioning of variables into subsets, which can overlap. Then SGPlan uses *constraint resolution* to resolve a) inconsistent variables falling into more than one subset and b) violated constraints which include variables from more than one subset. The subproblems are then solved by the *subproblem solver* – a modified Metric-FF planner [27].

TFD is a heuristic forward chaining planner which uses the context-enhanced additive heuristic [23] adapted to temporal and numeric planning.

POPF is a forward search planner that achieves to use some benefits of partial-order plan construction. It tries to find a compromise between least-commitment used in the partial-order planning and total commitment used in the forward search planning.

5.3 Transformation of the Requirements to PDDL

In order to get a valid input for the planners, we at first have to formalize a designer's requirements a little. Then we have to translate them to PDDL. We will now present the mechanism of this conversion and demonstrate it on a simplified example. The example we use is a soldier living in a virtual world of a fantasy RPG game. We will start by the description of this virtual world from the point of a designer's requirements. It will be followed by the requirements on the history of the soldier.

The virtual world. A designer wants the virtual world to contain ten cities and one village. A character living in this world should be able to, among other things, become a soldier by studying a military academy. Then he can take part in battles which happen in the period in question (each of them in a particular moment). The designer has an approximate idea about the total number of soldiers involved in each battle. She wants to preserve the relative rate of participation in the battles among the agents to be generated. So each battle has a probability assigned which expresses the chance a soldier will take part in it. A character in the world should be also able to have relationships with other characters, meet them, date, get married etc. He can also earn, find or win some money in various ways.

The character. A designer wants to generate the contents of episodic memory of a man named John, who is about 16 years old in the beginning of the period to be generated. He lives in the village and his cash consists of ten golden coins. About four years later, when this character is supposed to meet the player's character, the designer wants John to be married (so we have to include a woman in the setting, we will call her Stacy). He should become a soldier and take part in the battle for City1. He should also find a lot of money. And there should be included some random actions in his history, to make it more interesting.

In Chapter 4, we presented the types or requirements which a designer can have on the history of an agent. One of the goals of this work was to find a mechanism to convert these requirements to a valid input for planner, i.e. to PDDL. We will now describe this mechanism using the example defined above. For explanatory purposes, we will use an extract of our testing domain which is depicted in Fig. 5.1 and the extract from the problem depicted in Fig. 5.2. We will be referencing to the numbered parts of the pictures in the text.

Requirements on the Virtual World

The logical representation of the actions possible in the world are PDDL's durative actions. The locations and objects always present in the world can be translated to constants (D1 – Fig. 5.1). However, the objects which are present in the world only in this particular task should be translated to objects (P1 – Fig. 5.2).

Requirements on an Agent

1. **General requirements.** An example of this requirement is that the designer wants John to be a married soldier. To be married or to be a soldier are

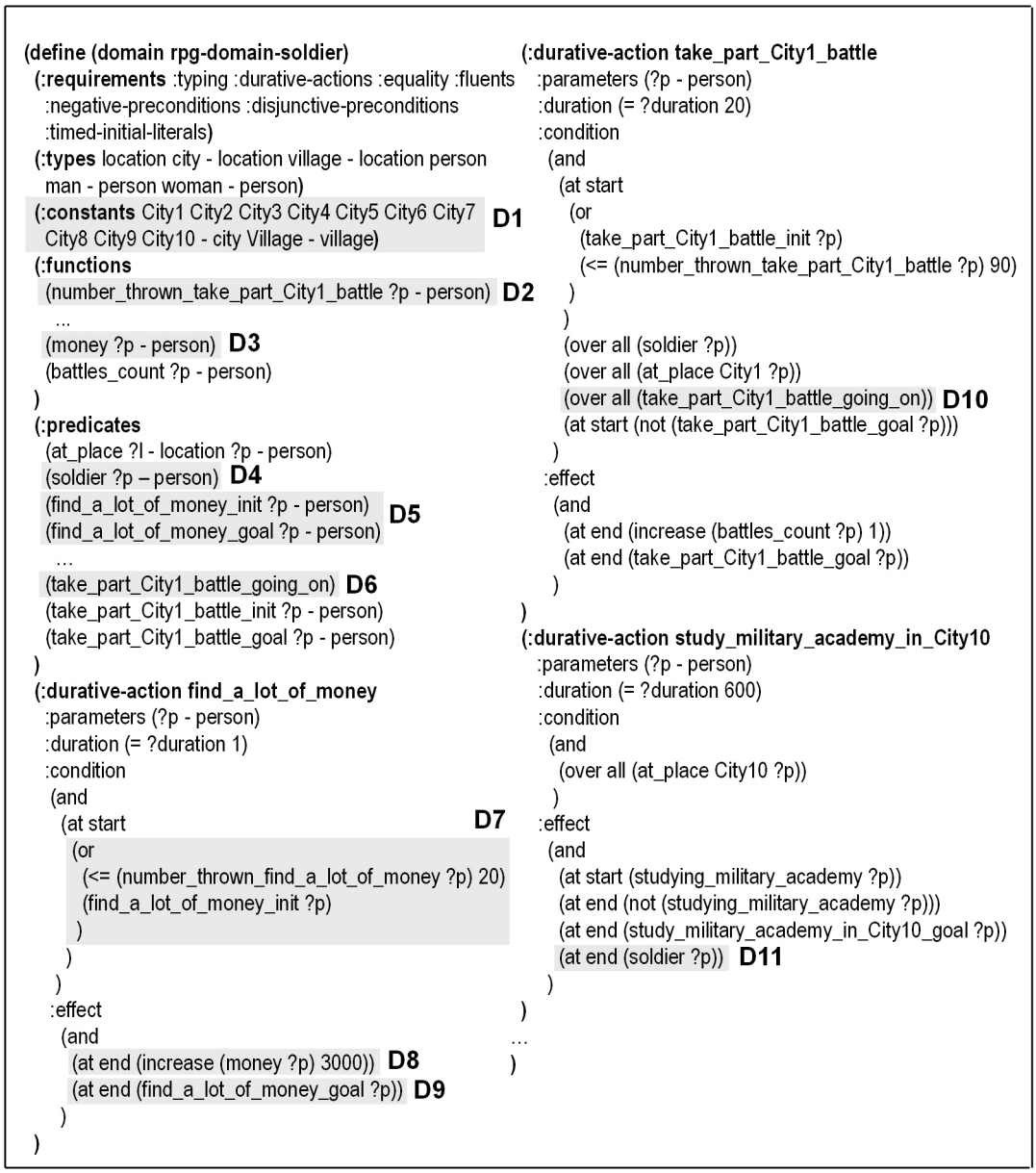


Figure 5.1: An extract from the planning domain. Description in the text.

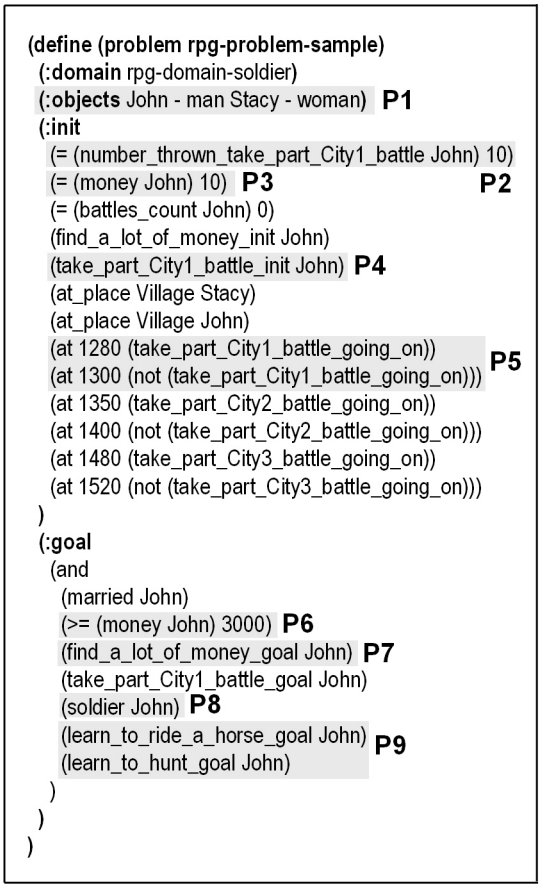


Figure 5.2: Sample PDDL problem. Description in the text.

logical true/false states of a person. Thus we can convert these requirements to PDDL predicates with a parameter of type person (D4). There are also numerical requirements in this category, e.g. we could want a character to have certain amount of money without caring how did he get them. These requirements can be translated to numeric fluents (D3). The predicates (or changes of the fluents) then become effects of one or multiple actions (D11, D8). In the definition of the PDDL problem, we have to list the predicates in the :init or :goal section if we want them to be true in that moment (P8) – if a predicate is not declared to be true, it is treated as false. We always have to set the initial value of the numeric fluents used in the domain, to avoid changes of undefined fluents (P3). When we have some requirements on the value of the fluent in the end state, we also have to include them in the :goal section (P6). We included it to our sample problem only for demonstration of use, it is not really necessary here to fulfill the requirements of the model example.

2. **Requirements on a concrete action.** In our example, the designer wants John to find a lot of money, i.e. we need to achieve that this particular action will appear in the resulting plan. This can be forced by adding a predicate (D5) which is set true after performing this action (D9) and in no other case. If we then insert this predicate in the :goal section of the problem definition (P7), all the valid plans have to include this action. In fact, we have to insert another predicate (D5) if this action has a probability associated. The reason for this will be explained in Req. 4a.

3. **Requirements on an action in a concrete time.** John should take part in the battle for City1. However, this action cannot be performed in any moment, John can only take part in it when it is going on. Here we can advantageously use timed initial literals – predicates, which are set to true/false automatically in a given moment. We can insert another predicate to the domain (D6) and make it a necessary precondition of the action in question during all the time of its execution (D10). In the problem definition, we then specify when this predicates turns true/false (P5).

4. **Randomness.**

   (a) **Probability of the actions.** In our virtual world, participations in different battles have different probabilities. The designer may want a character to take part in at least two battles, not caring which concrete battles these will be. She may just want to preserve the relative rates of soldiers taking part in each of the battles. Probability of an action is a numeric value, so we resolve this requirement by inserting a numeric fluent with a parameter of type person for each action with probability (D2). This numeric fluent is set to a random value for each person in the problem definition (P2). And a value small enough of this fluent is required in the action definition to allow the action to be performed (D7). However, the designer may want a character to perform an action of this type without regarding its probability. In our example, the designer wants John to take part in the battle for City1. This is why we include another

predicate for this type of actions (D9) which can override probability (D7). In this case, we set the predicate to true in the :init section of the problem definition (P4).

(b) **Insignificant actions to "animate" a virtual agent.** To force a planner to include a random action from a set of actions to the plan, we could insert to the domain a predicate which is set to true as an effect of the actions from this set (and only by them). Then we could require this predicate to be true in the goal state. Nevertheless, we could not specify in this way that we want the planner to include to the resulting plan more than one action from this set. Moreover, we would not be able to control the frequency of insertions of particular actions, although we may have an idea that some "noise actions" are more probable than other. A general-purpose planner would probably always insert to the plan the first action from the set. So we decided that if we include a preprocessing step in our design method, the best way to achieve inserting insignificant actions to the plan is to do it in this step. The designer then can simply tell the authoring tool how many actions marked as "noise actions" she wants to include and the tool will add predicates to the goal state (and the initial state, if necessary) analogically as in Req. 2 (P9).

5.4 High-level Language and Its Transformation to PDDL

As shown in the previous section, the transformation of some requirements to PDDL is not very straightforward. To define that a concrete action has to happen during the period to be planned (Req. 2), we have to introduce an artificial predicate to the domain and add it to the action's affects. Allowing an action to be undertaken only in a concrete time (Req. 3) has to be specified using timed initial literals, which turns to be difficult to manage when we have a lot of these actions in our domain. PDDL also has no structure to introduce probability of actions (Req. 4a), this has to be done by introducing artificial functions into the domain and using them in preconditions of an action. And above all, introducing some random "noise actions" (Req. 4b) to the history cannot be done directly in PDDL, we have to do this during a preprocessing step. For these reasons, we think a designer should be provided with a high-level formalism better suited for specifying her requirements. We will now introduce this formalism here and show how it can be converted to PDDL automatically, avoiding so a lot of repetitive work and errors.

Input Definition

We will now define formally the structure of the high-level designer language. It is not necessary for our present purposes to define exact syntax of the language. It is also possible that it will not be necessary ever, as a user-friendly authoring tool should enable the designer to input his requirements in an intuitive way, by filling in forms, for example. So we will describe just the abstract structure of the language, together with notes and examples to ease its comprehension to the reader. When it

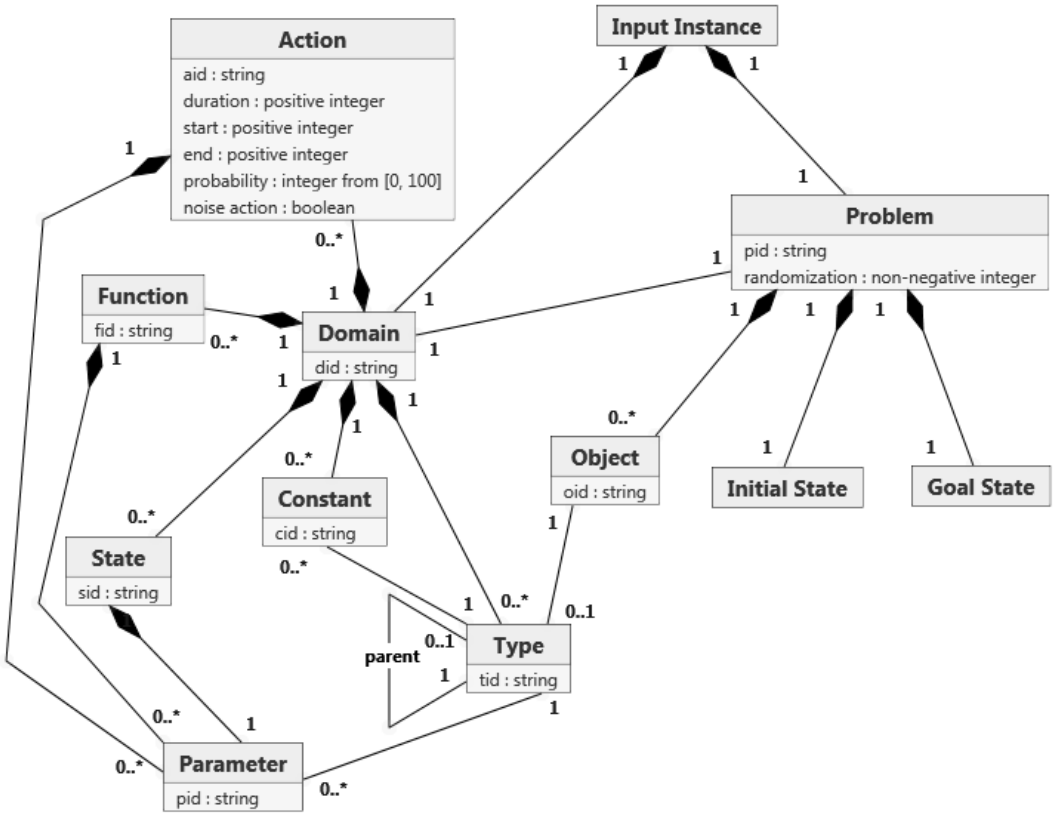


Figure 5.3: Graphical depiction of an *input instance*. See the text for further description.

is convenient, we will use the example of a soldier in a fantasy RPG. We also include graphical schemes of the structure in UML.

Definition 1 Input instance An *input instance* is a pair

$$Inst = (Dom, Prob)$$

where Dom is a domain and $Prob$ is a problem.

Input instance represents all the designer's requirements, both on the virtual world and the agents for whom the history is to be generated. The graphical structure of an input instance is depicted in Fig. 5.3.

Definition 2 Domain A *domain* is a 6-tuple

$$Dom = (did, T, C, F, S, A)$$

where did is the domain identifier, T is a list of types, C is a list of constants, F is a list of functions, S is a list of states and A is a list of actions.

A domain is a representation of a virtual world, represents its topology, actions, states which can be adopted by an agent etc.

Definition 3 Type A *type* is a pair

$$Type = (tid,\ p)$$

where *tid* is a unique identifier of the type and p is an identifier of a defined parent type - the type from which the new type inherits. The parent type can be null.

Some actions and states are logically possible just for some types of objects. For example, only a person can get married, not a location. That is why we need a system of types. There are several types which are predefined in our system - location, person, man and woman (the last two inherit from person).

Definition 4 Constant A *constant* is a pair

$$Constant = (cid,\ t)$$

where *cid* is a unique identifier of the constant and t is an identifier of a defined type or null for untyped constants.

A constant represents a part of the world which can take part in an agent's actions. A typical constant is a city or another location in the world.

Definition 5 Function A *function* is a pair

$$Function = (fid,\ P)$$

where *fid* is a unique identifier of the function and P is a list of its parameters. It represents an integer value.

Generally, a function is a parametrized variable with a numeric value. For instance, it describes how much money has a particular agent or in how many battles has he taken part.

Definition 6 Parameter A *parameter* is a pair

$$Param = (pid,\ t)$$

where *pid* is an identifier of a parameter (unique in a definition of one function, state or action) and t is an identifier of a defined type or null for parameters which can be of any type.

Definition 7 State A *state* is a pair

$$State = (sid,\ P)$$

where *sid* is a unique identifier of the state and P is a list of its parameters. It represents a boolean value.

We need states to describe true/false states of the world and the agents, e.g. whether an agent is married, whether he is a soldier, whether he can ride a horse etc.

Definition 8 Action An *action* is a tuple

$$Action = (aid,\ P,\ d,\ s,\ e,\ pb,\ ns,\ TC,\ E)$$

where *aid* is a unique identifier of the action, P is a list of its parameters, d is its duration in time steps (a positive integer), s is the first time step when an agent can start this action, e is the last time step when an agent can end this action (last two are positive integers or nulls for actions which can take place in any moment), pb is its probability (null if this action does not need probability), ns is true if this is a "noise action" (int this case, P can contain only one parameter, which is always of type person) and false if not, TC is a list of timed conditions describing necessary preconditions which has to be fulfilled for the action to take place and E is a set of effects of the action.

This is the key part of the description of a virtual world, it describes actions which are possible in it. As we are focusing on actions which can be undertaken by a human-like virtual agent, the list of parameters of every action must include at least one parameter of type person. The definition of an action also includes its probability and the time window when the action can happen, if necessary. For a designer, this is arguably better way to handle with these aspects than manually introducing artificial fluents and predicates directly into PDDL.

The graphical structure of an action is depicted in Fig. 5.4.

Definition 9 Effect An *effect* is a pair

$$Effect = (tv,\ ae)$$

where tv is the time of validity of the effect, i.e. the moment in the execution of the action when this effect takes place (its possible values are "start" and "end") and ae is an atomic effect, which can be an instantiated state, a negated instantiated state or a function change.

If an agent studies a military academy, in the end he becomes a soldier. When this soldier wins some money in poker, one of the effects of this action is that the amount of his money rises. But an effect also can become true at the beginning of the action, for example, when an agent starts moving from one location to another, he is no more in the original location.

Definition 10 Instantiated state An *instantiated state* is a pair

$$IS = (sid,\ CP)$$

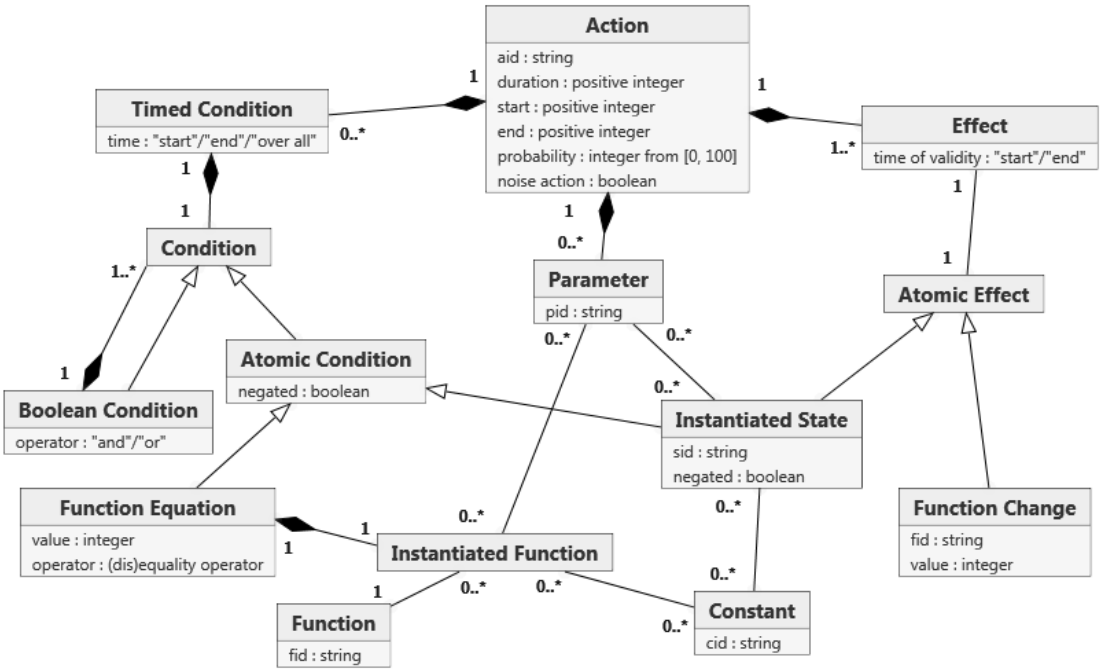


Figure 5.4: Graphical depiction of an *action*. See the text for further description.

where *sid* is an identifier of a defined state and *CP* is a list of defined constant identifiers and parameter identifiers defined in the current action. The number and types of the constants and parameters have to correspond to the state with identifier *sid*. An instantiated state has a boolean value.

An instantiated state is a concrete instance of a state, i.e. all his parameters have assigned concrete values (parameters defined in the action in which this instantiated state is used or constants defined in the domain). When used as an effect of an action, this instance becomes true in the time of validity of the effect. When used as a condition of an action, this instance must be true in the time of validity of this condition.

Definition 11 Function change A *function change* is a triple

$$FC = (fid,\ CP,\ val)$$

where *fid* is an identifier of a defined function, *CP* is a list of defined constant identifiers and parameter identifiers defined in the current action and *val* is a value to be added to the actual function value (integer number). The number and types of the constants and parameters have to correspond to the function with identifier *fid*. As an effect of an action, describes the change of the value of the function in the point specified by the parameters.

A function change is one of possible effects of an action. For example, if we have defined function money_amount with one parameter of type person, representing

the amount of money an agent owns, then when an agent performs an action like win_some_money, money_amount of this agent increases.

Definition 12 Timed condition A *timed condition* is a pair

$$TC = (t,\ c)$$

where t defines validity of the condition ("start", "end", "over all") and c is a condition.

Basically, a timed condition is a condition of an action enriched with the information when this condition has to be fulfilled to allow the action to be undertaken – at the start of the action, at the end or during all the time when the action is executed.

Definition 13 Condition A *condition* is an atomic condition, a negated atomic condition or a boolean condition. It has a boolean value.

Definition 14 Atomic condition An *atomic condition* is an instantiated state, an instantiated action or a function equation.

Definition 15 Boolean condition A *boolean condition* is a pair

$$BC = (oper,\ C)$$

where *oper* is a boolean operator ("and" or "or") and C is a list of conditions.

Definition 16 Instantiated action An *instantiated action* is a pair

$$IA = (aid,\ CP)$$

where *aid* is an identifier of a defined action and CP is a list of defined constant identifiers and parameter identifiers defined in the action in whose definition this instantiated action is used. The number and types of the constants and parameters have to correspond to the action with identifier *aid*. It has a boolean value.

An instantiated action is analogical to an instantiated state. If an instantiated action IA is used as an atomic condition of an action A, it means that the action identified by *aid* must have happened earlier in the history, with according parameters, to allow the action A to be executed.

Definition 17 Function equation A *function equation* is a triple

$$FE = (IF,\ val,\ oper)$$

where IF is an instantiated function, *val* is a numeric value (integer) and *oper* is one of the operators =, <=, >= and !=. It has a boolean value, it is true when the (in)equation "value of *IF*" *oper val* holds true.

If a function equation is used as an atomic condition of an action, it means that the value instantiated function in question has to satisfy the (in)equation defined by *val* and *oper*.

Definition 18 Instantiated function An *instantiated function* is a pair

$$IF = (fid,\ CP)$$

where *fid* is an identifier of a defined function and CP is a list of constant identifiers defined in the domain and parameter identifiers defined in the action in whose definition this instantiated function is used. The number and types of the constants and parameters have to correspond to the function with identifier *fid*. It has an integer value.

Instantiated function is analogical to instantiated state, the only difference is that instantiated function has numeric and not boolean value.

Definition 19 Problem A *problem* is a 6-tuple

$$Prob = (pid,\ did,\ O,\ is,\ gs,\ ran)$$

where *pid* is the problem identifier, *did* is an identifier of a defined domain, O is a set of objects, *is* is an initial state, *gs* is a goal state and *ran* is the level of randomization of the problem (a number of "noise actions" per agent which should be included in the output).

A problem is a representation of the requirements on a virtual agent demanded by a designer. The graphical structure of a problem is depicted in Fig. 5.5.

Definition 20 Object An *object* is a pair

$$Object = (oid,\ t)$$

where *oid* is a unique identifier of the object and t is an identifier of a defined type or null for untyped objects.

The difference between constants and objects is that constants is that constants form a solid part of the virtual world in question while objects are typically important just for one task. A most typical example of an object in this sense is a virtual agent, but an object also can be a sword, a pot of gold etc.

Definition 21 Initial state An *initial state* is a pair

$$IS = (GFV,\ GS)$$

where GFV is a list of grounded function values and GS is a list of grounded states or negated grounded states.

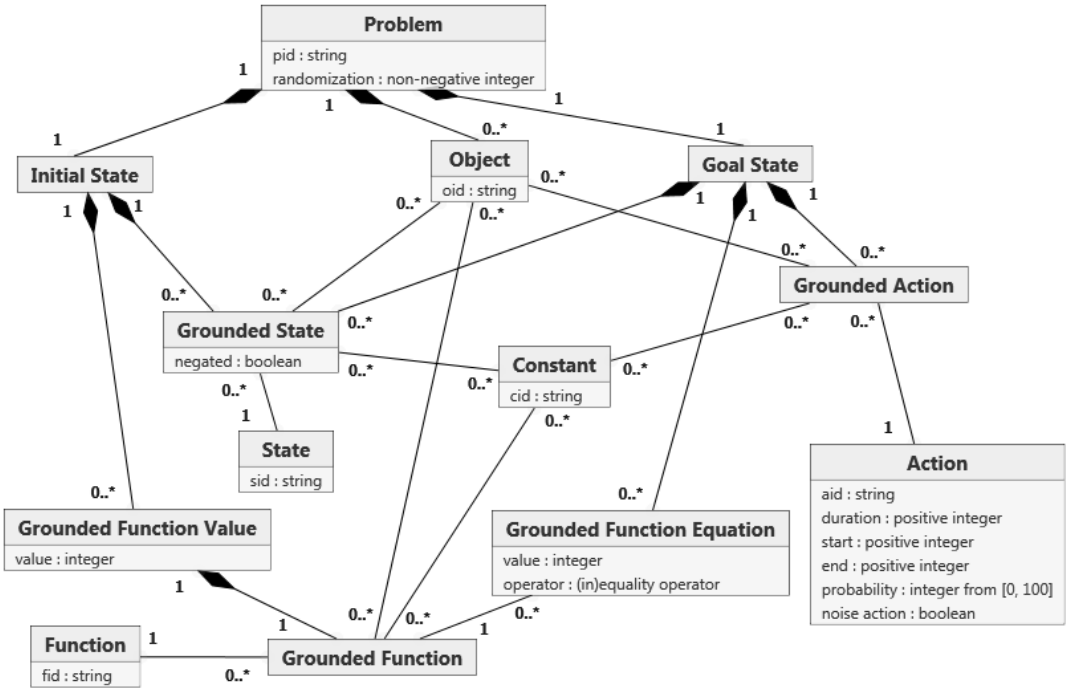


Figure 5.5: Graphical depiction of an *problem*. See the text for further description.

An initial state describes the state of the domain and the agents in the moment from which we want to start to generate the contents of episodic memory.

Definition 22 Grounded function value A *grounded function* value is a pair

$$GFV = (GF, val)$$

where GF is a grounded function and val is a numeric value (integer number).

A grounded function value defines the value of a grounded function in the initial state.

Definition 23 Grounded function A *grounded function* is a pair

$$GF = (fid, CO)$$

where *fid* is an identifier of a defined function and CO is a list of defined identifiers of constants and objects. The number and types of the constants and the objects have to correspond to the function with identifier *fid*.

The difference between an instantiated function and a grounded function is that an instantiated function is used inside a particular action in the domain definition, thus its parameters can be instantiated by the constants defined in the domain or by the parameters of the action in question. Meanwhile a grounded function is used

in the problem definition, so its parameters can be instantiated or by the constants or by the objects defined in the problem.

Definition 24 Grounded state A *grounded state* is a pair

$$GS = (sid,\ CO)$$

where *sid* is an identifier of a defined state and CO is a list of defined identifiers of constants and objects. The number and types of the constants and the objects have to correspond to the state with identifier *sid*.

A grounded state is analogical to a grounded function. When used in the definition of an initial state, it defines a state which is true in the moment from which we want to start to generate the contents of episodic memory. When used in the definition of a goal state, it defines a state which has to be true in the end of the generated period.

Definition 25 Goal state A *goal state* is a triple

$$GoS = (GS,\ GA,\ GFE)$$

where GS is a set of grounded states and/or negated grounded states, GA is a set of grounded actions and GFE is a set of grounded function equations.

A goal state describes the state of the domain and the agents which has to be reached by the end of the period to be generated.

Definition 26 Grounded action A *grounded action* is a pair

$$GA = (aid,\ CO)$$

where *aid* is an identifier of a defined action and CO is a list of defined identifiers of constants and objects. The number and types of the constants and the objects have to correspond to the action with identifier *aid*.

A grounded action is analogical to a grounded state. Using a grounded action in the definition of goal state means that the corresponding action has to be undertaken during the generated period.

Definition 27 Grounded function equation A *grounded function equation* is a triple

$$GFE = (GF,\ val,\ oper)$$

where GF is an grounded function, *val* is a numeric value (integer number) and *oper* is one of the operators $=$, $<=$, $>=$ and $!=$.

A grounded function equation defines the value of a grounded function which this grounded function must have in the end of the generated period.

Name: ConvertDomain
Input: Domain *Dom = (id, T, C, F, S, A)*
Output: A valid PDDL domain

writeline "(define (domain ", *id*, ")"
writeline "(:requirements :typing :durative-actions :equality :fluents :negative-preconditions :disjunctive-preconditions :timed-initial-literals)"
write "(:types "
write "location person man - person woman - person "
foreach type *Type = (tid, p)* **in** *T*
    write *tid*, " "
    if p != null write "- ", *p*, " "
writeline ")"
write "(:constants "
foreach constant *Constant = (cid, t)* **in** *C*
    write *cid*, " "
    if t is not null write "- ", *t*, " "
writeline ")"
ConvertFunctions(*F*, *A*)
ConvertPredicates(*S*, *A*)
ConvertActions(*A*)

Figure 5.6: Domain conversion algorithm.

Conversion to PDDL

We have showed how a designer's requirements can be expressed in PDDL. We have also defined the structure of a high-level language which is better suited to formulate these requirements. Now we will describe how this high-level input can be translated to PDDL automatically.

Generally, a domain is converted to a PDDL domain and a problem is converted to a PDDL problem, as depicted in Fig. 5.6 and 5.7, respectively.

When translating a domain, types, constants and functions are converted directly to PDDL types, constants and numeric fluents. There are also added predefined types mentioned in the previous section and one numeric fluent for each action with probability (Fig. 5.8). States are converted to PDDL predicates and actions to PDDL durative actions.

We also generate from one to three predicates for each action, these are:

- *aid_goal* – used to enable the designer to specify a concrete action for an agent, as it is also added to the list of effects of the action in question.
- *aid_init* – used for actions with probability, to overdrive it in case when the designer wants the particular action to be included in output.
- *aid_going_on* – used for actions which can occur just in a specified time window, the validity of this predicate is added to the list of conditions of the action.

The preconditions of the generated durative actions are a direct reflection of conditions of actions, there are just added some new preconditions in case of actions with

Name: ConvertProblem
Input: Domain *Dom = (did, T, C, F, S, A)* and problem *Prob = (id, did, O, is, gs, ran)*
Output: A valid PDDL problem

writeline "(define (problem ", *id*, ")"
writeline "(:domain ", *did*, ")"
write "(:objects "
foreach object *Object = (oid, t)* **in** O
    write *oid*, " "
    if t is not null write "- ", *t*, " "
writeline ")"
NA := list of pairs *P = (random noise action identifier, person identifier)*, where there are *ran* actions for every *person* in *O*
ConvertInitState(*is*, *gs*, *NA*, *O*, *A*)
ConvertGoalState(*gs*, *NA*, *A*)

Figure 5.7: Problem conversion algorithm.

Name: ConvertFunctions
Input: Set of functions *F*, set of actions *A*
Output: Definition of PDDL fluents belonging to the domain

writeline "(:functions"
foreach action *Action = (aid, P, d, s, e, pb, ns, c, E)* **in** *A* **where** *pb* != null
    writeline "number_thrown_", *aid*, "?p - person"
foreach function *Function = (fid, P)* **in** *F*
    write *fid*
    ConvertParameters(*P*)
writeline ")"

Figure 5.8: Algorithm to convert functions.

Name: ConvertActionCondition
Input: Action *Action = (aid, P, d, s, e, pb, ns, TC, E)*
Output: Definition of PDDL condition of a durative action

writeline ":condition"
writeline "(and"
if *pb* != null
    writeline "(at start"
    writeline "(or"
    writeline "(<= (number_thrown_", *aid*, " ?p) ", *pb*, ")"
    writeline "(", *aid*, " _init)"
    writeline ")"
    writeline ")"
if s != null writeline "(over all (", *aid*, "going_on))"
foreach *TCond = (t, c)* **in** *TC*
    if t == "start" writeline "(at start"
    elseif t == "end" writeline "(at end"
    else writeline "(over all "
    ConvertCondition(*c*)
    writeline ")"
writeline ")"

Figure 5.9: Algorithm to convert conditions of actions.

probability or actions which can occur only in a specified time window, as depicted in Fig. 5.9. Predicate *aid_ goal* is added to the effects of each action, which is later used for problem generation.

When translating a problem, objects are converted to PDDL objects. We include to the :init section of the problem definition direct translation of initial state specified by the designer, as well as the definitions of time windows for the actions (timed initial literals) and random values of fluents included for probability (Fig. 5.10). We also choose a specified number of noise actions per agent and include predicates of form *aid_ init* to to :init section, if this noise actions have probability - in that case, we have to override it. Generating the :goal section of the problem definition is very straightforward, similar to the generation of the :init section.

For readability, we include in this chapter only algorithms mentioned in the text above. The rest of algorithms needed for the conversion can be found in Appendix B.

5.5 Summary

In this chapter, we have summarized necessary features of a planner suitable for our task. We have chosen three general-purpose planners which fulfill these criteria. Then we have presented a mechanism how a designer's requirements on a virtual world and the history of an agent can be translated to PDDL. Stemming from the mechanism, we have introduced the structure of a high-level language which is better suited for specifying these requirements. Then we have showed how this language can be automatically converted to PDDL. By doing so, we have fulfilled the first and partly the second goal of this work as listed at the end of Chapter 1.

Name: ConvertInitState
Input: Initial state *IS = (GFV, GS)*, goal state *GoS = (GSt, GA, GFE)*, list of noise actions and person identifiers *NA*, list of objects *O*, list of actions *A*
Output: A valid PDDL definition of initial state

writeline “(:init”
foreach action *Action = (aid, P, d, s, e, pb, ns, c, E)* **in** A
    if *s* != null
        writeline “(at ”, *s*, “ (”, *aid*, “ _going_on))”
        writeline “(at ”, *e*, “ (not (”, *aid*, “ _going_on)))”
    if *pb* != null
        foreach object of type person *Object = (oid, t)* **in** *O*
            writeline “(= (number_thrown_”, *aid*, “ ”, *oid*, “) ”,
                random value from [0, 100], “)”
foreach grounded function value *GFVal = (GF, val)* **in** *GFV*
    GF = (fid, CO)
    write “(= (”, *fid*
    foreach constant or object identifier *id* **in** *CO*
        write “ ”, *id*
    writeline “) ”, *val*, “)”
foreach grounded state *GState = (sid, CO)* **in** *GS*
    write “(”, *sid*
    foreach constant or object identifier *id* **in** *CO*
        write “ ”, *id*
    writeline “)”
foreach grounded action *GAction = (aid, CO)* **in** *GA* **where** the maternal
        action’s *pb* != null
    write “(”, *aid*, “ _init”
    foreach constant or object identifier *id* **in** *CO*
        write “ ”, *id*
    writeline “)”
foreach *P = (aid, pid)* **in** *NA* **where** the maternal action’s *pb* != null
    writeline “(”, *aid*, “ _init ”, *pid*, “)”
writeline “)”

Figure 5.10: Algorithm to convert initial state.

Chapter 6

Results

In this chapter, we will first present the testing environment. We will discuss the design of the experiments and describe the planning domain we created. This will be followed by the list of the planners and hardware used in our tests. Then we will describe closely our experiments, present their results and discuss them.

6.1 Testing Environment

Design of the Experiments

The design of our experiments stemmed from the fact that the task is scalable in several dimensions:

1. The number of actions which must be included in a plan to solve the problem.
2. The number of agents included in the problem.
3. The amount and complexity of interconnections between agents.
4. The type of the actions that must be included in the plan to solve the problem – whether their execution is limited only to certain time windows.
5. The size of the domain.

Points 1 and 2 are similar, at least when we are speaking about ten or twenty agents as a maximum. So we decided to treat them as a single criterion during the design of our experiments. To test the influence of the movement along the resulting four scales, we created four experiments featuring standalone agents and four experiments featuring agents with relationships between them (point 3). In each group of experiments, there are two experiments using actions limited to certain time windows and two that do not use them (point 4). One of this pair of experiments is performed on the complete domain, one on a reduced one (point 5). And finally, each experiment is composed from several tests which have growing number of included actions in case of standalone agents (points 1 and 2) or growing complexity of relationships in case of interconnected agents (point 3).

Testing Domain

To test our approach, we manually created a PDDL domain according to the algorithms described in the previous chapter. This domain describes a simplified fantasy virtual world inspired by RPG games. The world contains ten cities and one village. The actions and states of an agent possible in this world could be logically divided in these seven categories: *basic*, *soldier*, *mage*, *studies*, *relationships*, *money* and *other*. We will present them more in detail now.

Basic. This group serves for describing location and movement of the agents in the world. Contains the predicate *at_ place ?l - location ?p - person* and the action *move* with two parameters of type location and one parameter of type person. The duration of this action is always 3 (meaning three days), as we actually want to focus on other aspects than the exact topology of the world.

Soldier. This category contains above all predicates expressing whether an agent is a soldier and which is his rank. There are several predicates of types *_ init*, *_ going_ on* and *_ goal*, needed for Req. 2, 3 and 4, as described in the previous chapter (this is also true for the rest of the categories, we will not mention them repeatedly in their descriptions). The actions in this group can be divided into three subgroups:

- Studying a military academy - *study_ military_ academy_ in_ City10 ?p - person*. This is a long durative action. After studying the military academy, an agent becomes a soldier.
- Battles. Ten actions like *take_ part_ in_ City1_ battle ?p - person*, one for each city. These are durative actions lasting several days which can only occur in a specified interval (time window). The actions have probability, so there are also defined functions like *number_ thrown_ take_ part_ City1_ battle ?p - person*. An agent can take part in a battle only if the value of the corresponding function is low enough. Or when his participation in a given battle is required directly in the goal.
- Promotions to higher ranks. After taking part in given number of battles, monitored by the function *battles_ count ?p - person*, a soldier can be promoted to lieutenant, captain etc. These actions are short.

Mage. The structure of this category is similar to the previous one, but this time we describe mages. To become a mage, an agent has to study a magic university to master the basics of magic. Then he can start to learn spells (e.g. fireball, freezing, enchanting), these are also long actions lasting weeks or months. Each spell has four levels, after learning the first level of a spell, a mage can learn the second level of this spell and so on. The number of first level spells learned is described by functions *level1_ spells_ count ?p - person* and analogically for the rest of levels. After learning the first level of at least three spells, a mage can get the first degree of initiation, then the second level of initiation after learning the second level of at least three spells etc. - it is a variance on the promotions to higher ranks from the previous category. A mage can also attend magic conferences, there are five defined in the domain, each of them can take part only in a concrete time window.

Studies. This group describes general academic studies of an agent or skills that can be learned. It contains actions like studying the university or learning to ride a horse.

Relationships. This category represents common human relationships. We have here predicates like *have_met ?p1 - person ?p2 - person*, *married ?p - person*, *are_married ?m - man ?w - woman* etc. The actions in this group are meeting, dating, getting married, getting divorced and having a love affair.

Money. This category contains the function *money ?p - person*, describing the actual amount of money of an agent. There are also several actions which increase the value of this function as an effect, for example *win_some_money_in_poker* or *earn_some_money*.

Other. This group contains noise actions which do not fit to any of the previous group, e.g. *go_to_a_theater* or *meet_a_famous_hero*.

The resulting complete domain contains 11 constants, 26 numeric fluents with one parameter of type *person*, 132 predicates with one to three parameters and 73 durative actions with one to three parameters. However, in several tests we used smaller domains which do not involve all the mentioned packages, as described later.

Planners and Hardware

We had some issues with finding planners that would support all levels of PDDL 2.2, which is the version that supports all our requirements on PDDL, as described in the previous chapter. Generally, there is not a lot of such planners and many of them contain significant bugs and/or are several years old and no longer supported. We were able to find only one usable planner fully fulfilling our requirements. This planner is SGPlan6 [28].

To perform the tests using more than one planner, we had to yield some technical requirements, in concrete :timed-initial-literals to use TFD [16] and :timed-initial-literals, :negative-preconditions and :disjunctive-preconditions to use POPF [8]. To create domains equaling our original domain, but without need for these requirements, we used procedures described in 5.1. We also had to set the duration of all actions to at least 1 in order to use POPF, because POPF considers invalid the domains containing durative actions with zero duration.

The experiments were run on two PCs, one of them with Intel Core i5 2.66 GHz CPU and 2 GB RAM running Ubuntu 9.10 (PC1) and the second with Intel Core2 Quad 2.83 GHz CPU and 3 GB RAM running Gentoo Linux 10.1 (PC2).

The sources and/or binaries of the chosen planners, the domains and problems used in all the experiments as well as the resulting plans can be found on the accompanying CD in the corresponding directories.

6.2 Standalone Agents

Experiment 1: Standalone Mages

We designed an experiment with linearly increasing number of uniform agents who have no relationships between them. We wanted to see whether in this setting the computation time will also increase linearly.

Hypothesis. When generating contents of episodic memory for uniform agents who are not interconnected, the time required for computation will be growing approximately linearly with the growing number of agents.

Method. In this experiment, we are generating history of mages who are forced by the goal to attend the magic university and learn spells to get the third level of initiation. This setting does not need the planner to include in the plan actions that are limited to certain time windows (e.g. to attend a particular magic conference which took place at a given time in the history). An extract from the PDDL problem for generating contents of episodic memory of one mage (created using algorithms described in the previous chapter) is depicted in Fig. 6.1. PDDL for more mages is analogical.

We used the full testing domain as described in the previous section. When generating one, two resp. three agents, the task contains 252, 489 resp. 726 grounded predicates and so on. This is true for the domain and problems used for SGPlan6, the adjusted inputs for TFD and POPF contain more grounded predicates.

We ran the experiment with each planner for the number of mages from 1 to 20, each task with the deadline of 30 minutes. On both testing PCs, we performed five repetitions of each task.

Results. The average computation times are graphed in Fig. 6.2. The exact numbers, together with standard deviations, are listed in Table 6.1. As can be observed, the time needed for computation increases quickly and TFD and POPF did not accomplish to generate a plan for more than 4 resp. 7 agents, although the time needed to output plans for these number of agents was very short. The planner which performed the best in this experiment, SGPlan6, was able to find a plan for the maximum of 15 agents. In cases of more agents, the planners usually did not manage to find a plan in the limit of 30 minutes or, more frequently, exhausted all the available memory.

Discussion. Although one could expect that the time needed for computation will grow linearly or only a bit slower with increasing number of standalone agents of the same type, our results show that this hypothesis is not true for the used sample of state of the art general-purpose planners. Their heuristics arguably are not able to detect the isolation of the subgoals.

Experiment 2: Standalone Mages on a Smaller Domain

As the results of Experiment 1 were not very encouraging, we wanted to test the performance of the planners on a smaller domain.

```
(define (problem rpg-problem-exp1-liter1)
    (:domain rpg-domain-complete-liter)
    (:objects Man1 - man)
    (:init
        (at 1280 (take_part_City1_battle_going_on))
        (at 1300 (not (take_part_City1_battle_going_on)))
        (at 1350 (take_part_City2_battle_going_on))
        (at 1400 (not (take_part_City2_battle_going_on)))
        ...
        (= (number_thrown_find_a_lot_of_money Man1) 1)
        (= (number_thrown_earn_a_lot_of_money Man1) 1)
        ...
        (= (number_thrown_take_part_City1_battle Man1) 1)
        (= (number_thrown_take_part_City2_battle Man1) 1)
        ...
        (= (battles_count Man1) 0)
        (= (money Man1) 0)
        (= (level1_spells_count Man1) 0)
        (= (level2_spells_count Man1) 0)
        (= (level3_spells_count Man1) 0)
        (= (level4_spells_count Man1) 0)
        (at_place City1 Man1)
    )
    (:goal
        (and
            (get_third_degree_of_initiation_goal Man1)
        )
    )
)
```

Figure 6.1: Experiment 1 - extract of the PDDL problem.

	SGPlan6				TFD				POPF			
	PC1		PC2		PC1		PC2		PC1		PC2	
No. of agents	M	SD	M	SD	M	SD	M	SD	M	SD	M	SD
1	0.09	0.01	0.10	0.01	0.51	0.02	0.61	0.00	0.24	0.01	0.23	0.01
2	0.23	0.01	0.24	0.01	-	-	-	-	0.69	0.12	0.64	0.00
3	0.65	0.01	0.63	0.00	5.31	0.02	7.08	0.01	1.72	0.06	1.68	0.00
4	1.58	0.00	1.44	0.00	7.16	0.13	9.42	0.01	4.26	0.04	4.20	0.01
5	3.34	0.02	2.82	0.01	-	-	-	-	9.98	0.04	9.85	0.02
6	6.02	0.04	5.12	0.03	-	-	-	-	21.94	0.12	21.78	0.04
7	10.34	0.02	8.56	0.02	-	-	-	-	44.71	0.22	44.84	0.09
8	16.27	0.23	14.22	1.11	-	-	-	-	-	-	-	-
9	24.21	0.10	20.83	0.11	-	-	-	-	-	-	-	-
10	34.99	0.08	30.62	0.19	-	-	-	-	-	-	-	-
11	48.91	0.25	42.95	0.13	-	-	-	-	-	-	-	-
12	67.17	0.33	58.11	0.13	-	-	-	-	-	-	-	-
13	89.15	0.26	77.97	0.76	-	-	-	-	-	-	-	-
14	116.40	0.35	100.56	0.48	-	-	-	-	-	-	-	-
15	-	-	143.30	1.51	-	-	-	-	-	-	-	-

Table 6.1: Experiment 1 - times of computation (M - mean, SD - standard deviation).

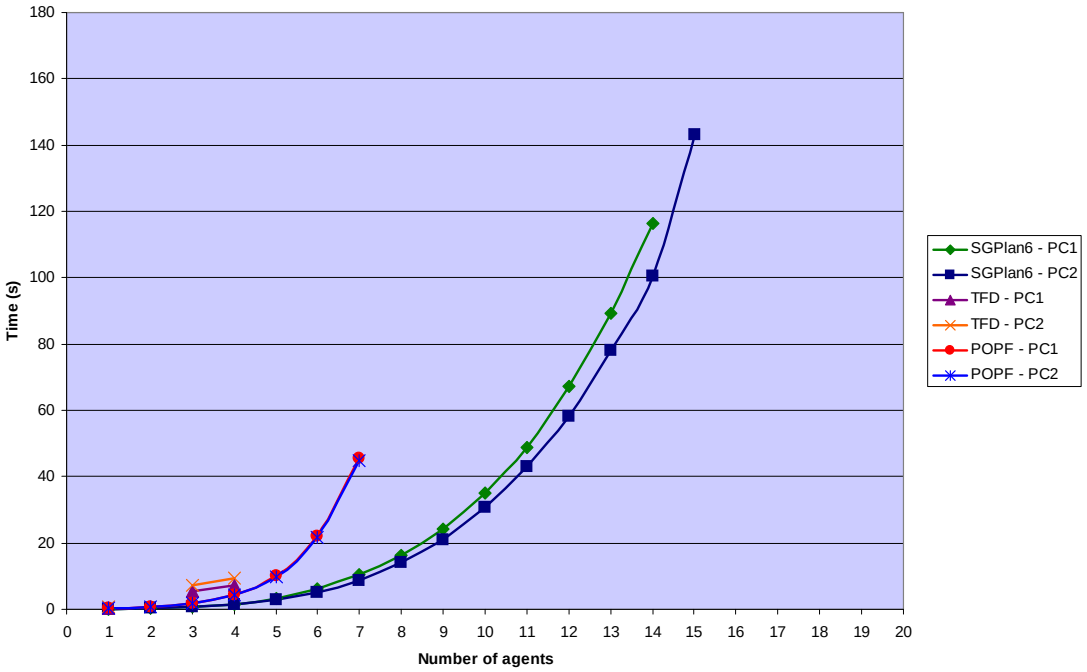


Figure 6.2: Experiment 1: Generating history of standalone mages - average times of computation.

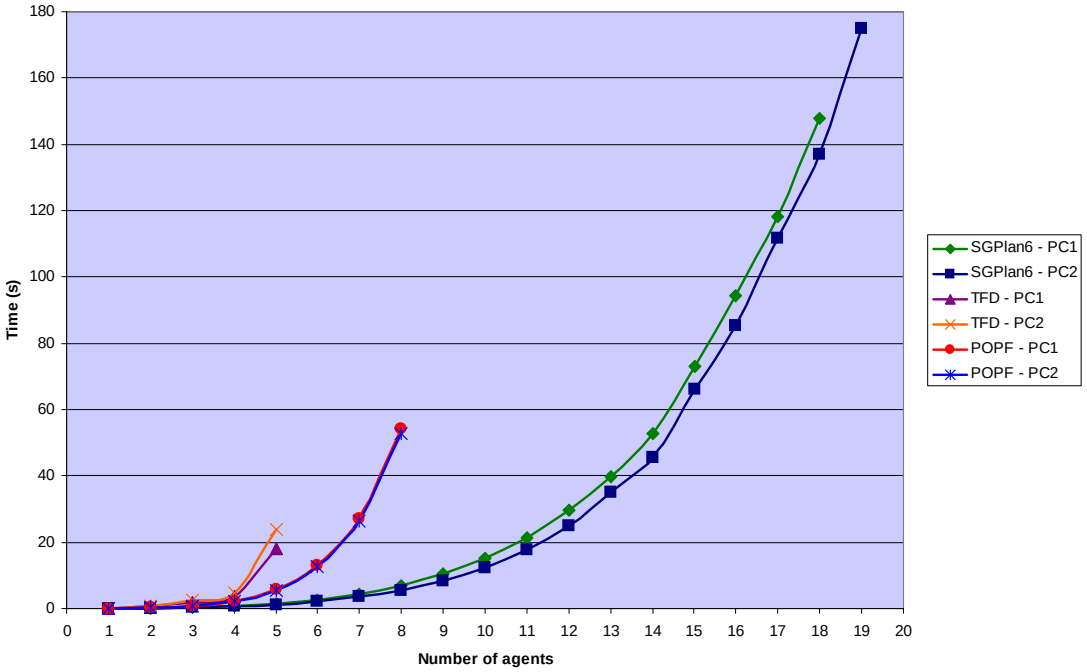


Figure 6.3: Experiment 2: Generating history of standalone mages on a smaller domain - average times of computation.

	SGPlan6				TFD				POPF			
	PC1		PC2		PC1		PC2		PC1		PC2	
No. of agents	M	SD	M	SD	M	SD	M	SD	M	SD	M	SD
1	0.05	0.00	0.04	0.00	0.11	0.01	0.12	0.00	0.10	0.01	0.08	0.00
2	0.10	0.01	0.09	0.00	0.54	0.01	0.67	0.00	0.29	0.01	0.28	0.00
3	0.26	0.01	0.23	0.00	1.88	0.02	2.39	0.00	0.85	0.01	0.85	0.00
4	0.61	0.01	0.56	0.00	3.77	0.02	4.83	0.01	2.29	0.01	2.28	0.00
5	1.34	0.01	1.15	0.01	18.12	0.04	23.76	0.03	5.65	0.13	5.55	0.00
6	2.52	0.01	2.09	0.00	-	-	-	-	12.91	0.41	12.50	0.01
7	4.31	0.02	3.47	0.00	-	-	-	-	27.09	0.25	26.25	0.00
8	6.91	0.14	5.53	0.01	-	-	-	-	54.06	0.75	52.88	0.11
9	10.32	0.07	8.40	0.02	-	-	-	-	-	-	-	-
10	15.10	0.05	12.30	0.03	-	-	-	-	-	-	-	-
11	21.29	0.07	17.82	0.03	-	-	-	-	-	-	-	-
12	29.51	0.12	24.99	0.07	-	-	-	-	-	-	-	-
13	39.71	0.11	34.90	1.24	-	-	-	-	-	-	-	-
14	52.88	0.09	45.64	0.15	-	-	-	-	-	-	-	-
15	72.84	0.20	66.29	0.39	-	-	-	-	-	-	-	-
16	94.30	0.25	85.21	0.82	-	-	-	-	-	-	-	-
17	118.32	0.22	111.80	4.03	-	-	-	-	-	-	-	-
18	147.83	0.49	137.09	4.06	-	-	-	-	-	-	-	-
19	-	-	174.99	3.56	-	-	-	-	-	-	-	-

Table 6.2: Experiment 2 - times of computation (M - mean, SD - standard deviation).

Hypothesis. By performing the experiments on a smaller domain, we will reduce the state space that has to be searched by the planner significantly. This will result in dramatically decreased times needed for computation of each task.

Method. We generated plans for the same PDDL problems as in Experiment 1 on a smaller domain including just packages *basic*, *mage* and *relationships*. When generating one, two resp. three agents, the task contains 178, 351 resp. 524 grounded predicates and so on. We performed five repetitions on each PC.

Results. The results of this test are graphed in Fig. 6.3. The exact numbers, together with standard deviations, are listed in Table 6.2. Using a smaller domain decreased the amount of time needed for computation significantly and postponed the problem of exhausting all the memory. As the result, SGPlan6 was able to output a plan for 19 mages, i.e. a plan containing as many as 264 actions.

Discussion. The results of this experiment support our hypothesis that reducing the domain could reduce required computation time a lot. This means that it could be very helpful to include a package managing system in the authoring tool. A designer will often know she does not need all the actions from the domain (if she is about to generate history of several mages, why to include actions which are typical only for a soldier). So she could use a package manager to include only packages she

```
(define (problem rpg-problem-exp3-liter1)
    (:domain rpg-domain-complete-liter)
    (:objects Man1 - man)
    (:init
        (at 1280 (take_part_City1_battle_going_on))
        (at 1300 (not (take_part_City1_battle_going_on)))
        ...
        (= (number_thrown_take_part_City1_battle Man1) 1)
        ...
        (= (battles_count Man1) 0)
        ...
        (at_place City1 Man1)
    )
    (:goal
        (and
            (>= (battles_count Man1) 2)
        )
    )
)
```

Figure 6.4: Experiment 3 - extract of the PDDL problem.

will need, and doing so probably reduce the time needed for computation. However, it does not solve efficiency issues completely, as a designer may need to generate history for several interconnected agents of different types. We can also anticipate other efficiency issues in the case when the actions needed to fulfill the goal can be executed only in a certain period of time, which is why we designed the following experiments.

6.3 Standalone Agents with Time Windows

Experiment 3: Standalone Soldiers

Hypothesis. If the goals in a PDDL problem require the planner to include actions which can be undertaken only in a specified time window, the complexity of the task for the planner will grow significantly.

Method. In this experiment, we also generated the contents of episodic memory for as many standalone agents as possible, but this time we were generating soldiers. These soldiers had to take part in at least two random battles. The main difference between the previous tests and this one is that a soldier can take part in each of the ten battles defined in the domain only in a certain time window. It stems from the fact that one can take part in a particular battle only when this battle takes place. A short extract from the PDDL problem for generating history of one soldier is depicted in Fig. 6.4. The tests were performed on the full testing domain. We ran the experiment with each planner for the number of soldiers from 1 to 10, each task with the deadline of 30 minutes. On both testing PCs, we performed five repetitions of each task.

Results. The results of this test were not very encouraging. Only the problem with one soldier was solved in the time limit and only by SGPlan6. This planner

	SGPlan6				TFD				POPF			
	PC1		PC2		PC1		PC2		PC1		PC2	
No. of agents	M	SD	M	SD	M	SD	M	SD	M	SD	M	SD
1	3.75	0.03	2.22	0.01	0.14	0.01	0.15	0.01	-	-	-	-
2	-	-	-	-	0.95	0.01	1.22	0.01	-	-	-	-
3	-	-	-	-	-	-	-	-	-	-	-	-

Table 6.3: Experiment 4 - times of computation (M - mean, SD - standard deviation).

needed to output a plan on average 821.33 s (standard deviation 0.44 s) on PC1 and 1469.46 s (standard deviation 29.28 s) on PC2, although this plan contains only six actions. TFD did not generate any output at all and POPF marked the problem as unsolvable.

Discussion. The results acknowledge our hypothesis that using actions which can occur only in certain time windows brings a lot of difficulties to the planners. The time limit used for the experiment, 30 minutes, is much shorter that would be a reasonable limit for real use, however, the PDDL problems used are also very simple. A designer's requirements in real use would be supposedly more complicated than just require one or two agents to take part in two battles. It suggests that for real application we would need a planner more optimized for planning with time windows.

Experiment 4: Standalone Soldiers on a Smaller Domain

The planners did not perform very well in the previous experiment. So we wanted to repeat it on a smaller domain, which could reduce the time of computation as in Experiment 2.

Hypothesis. By performing the experiments involving time windows on a smaller domain, we will reduce the space that has to be searched by the planner significantly. This will result in dramatically decreased times needed for computation of each task.

Method. We repeated the previous experiment on a smaller domain (Experiment 4) including only packages *basic*, *soldier* and *relationships*. When generating one, two resp. three agents, the task contains 179, 348 resp. 517 grounded predicates and so on. We performed five repetitions on each PC.

Results. The results are showed in Table 6.3. The time of computation needed by SGPlan6 to solve the problem with one soldier decreased dramatically, but despite of this the planner was not able to generate a plan for two or more soldiers in the limit. On the other hand, TFD achieved computing plans for one and two soldiers on this reduced domain, although it did not compute any plans on the complete domain.

Discussion. The results of the experiment support our hypothesis, although the performance of the planners was still unsatisfactory.

```
(define (problem rpg-problem-exp5-liter1)
    (:domain rpg-domain-complete-liter)
    (:objects John James - man Stacy - woman)
    (:init
        ...
    )
    (:goal
        (and
            (get_fourth_degree_of_initiation_goal John)
            (get_fourth_degree_of_initiation_goal James)
            (meet_goal John James)
            (married John)
        )
    )
)
```

Figure 6.5: Experiment 5, Part 1 - extract of the PDDL problem.

6.4 Interconnected Agents

After testing the generation of history of standalone agents, we also designed experiments with several interconnected agents to see how the relationships between agents influence the performance of the planners. We started by experiments without actions that can occur only in particular time windows.

Experiment 5: Interconnected Mages

Hypothesis. When there are relationships among the agents, i.e. when they are forced common points in their memories, the planners will have more difficulties with finding a valid plan. More relationships will need more computation time, as it introduces more constraints between plans for individual agents. Nevertheless, inserting only one or two common points to the memories of the agents should not complicate the task so much.

Method. We designed three tests that form parts of this experiment, with increasing complexity of connections between the agents. In the first part (Fig. 6.5), there are two men (John and James) who have to become skilled mages and meet. One of them also has to get married. In the second part, both of them have to get married (Fig. 6.6). And in the last test in this experiment, John is also supposed to have an affair with his friend's wife (Fig. 6.7). We used the full testing domain in the experiment. The task in part 1 contains 786 grounded predicates, in parts 2 and 3 there are 1143 grounded predicates. We ran the tests five times with each planner on both PCs. We used the time limit of one hour.

Results. The results are depicted in Table 6.4. SGPlan6 solved the first and the second problem very quickly, but it was not able to solve the last one. POPF was only able to solve the first problem, and only on one of the testing PCs, curiously. On the second one it exhausted all the available memory. TFD did not manage to solve any of the problems.

```
(define (problem rpg-problem-exp5-liter2)
    (:domain rpg-domain-complete-liter)
    (:objects John James - man Stacy Kate - woman)
    (:init
        ...
    )
    (:goal
        (and
            (get_fourth_degree_of_initiation_goal John)
            (get_fourth_degree_of_initiation_goal James)
            (meet_goal John James)
            (married John)
            (married James)
        )
    )
)
```

Figure 6.6: Experiment 5, Part 2 - extract of the PDDL problem.

```
(define (problem rpg-problem-exp5-liter3)
    (:domain rpg-domain-complete-liter)
    (:objects John James - man Stacy Kate - woman)
    (:init
        ...
    )
    (:goal
        (and
            (get_fourth_degree_of_initiation_goal John)
            (get_fourth_degree_of_initiation_goal James)
            (married John)
            (married James)
            (had_affair John Kate)
        )
    )
)
```

Figure 6.7: Experiment 5, Part 3 - extract of the PDDL problem.

	SGPlan6				TFD				POPF			
	PC1		PC2		PC1		PC2		PC1		PC2	
	M	SD	M	SD	M	SD	M	SD	M	SD	M	SD
Part 1	0.69	0.01	0.65	0.01	-	-	-	-	2.11	0.02	-	-
Part 2	2.51	0.01	2.22	0.01	-	-	-	-	-	-	-	-
Part 3	-	-	-	-	-	-	-	-	-	-	-	-

Table 6.4: Experiment 5: Interconnected agents without time windows. Times of computation in seconds (M - mean, SD - standard deviation).

	SGPlan6				TFD				POPF			
	PC1		PC2		PC1		PC2		PC1		PC2	
	M	SD	M	SD	M	SD	M	SD	M	SD	M	SD
Part 1	0.44	0.01	0.39	0.01	21.04	0.23	27.11	0.04	1.13	0.00	0.76	0.00
Part 2	0.89	0.00	0.80	0.01	-	-	-	-	5.06	0.02	4.96	0.01
Part 3	-	-	-	-	-	-	-	-	4.68	0.02	4.59	0.01

Table 6.5: Experiment 6: Interconnected agents without time windows on a smaller domain. Times of computation in seconds (M - mean, SD - standard deviation).

Discussion. The results suggest that relationships between agents increase the complexity of the task, as none of the planners was able to output a plan for the third problem, although there is a valid plan for this task with only 78 actions and the longest plan outputted by SGPlan6 in Experiment 1 was 208 actions long.

Experiment 6: Interconnected Mages on a Smaller Domain

As in the previous experiments, we wanted to determine the effect of reducing the testing domain on the performance of the planners.

Hypothesis. Reducing the domain will significantly decrease the difficulty of the tasks for the planners.

Method. We repeated the tests from the previous experiment on a smaller domain containing only packages *basic*, *mage* and *relationships*. The task in part 1 contains 584 grounded predicates, in parts 2 and 3 there are 817 grounded predicates. We again performed five repetitions of each test with each planner on both PCs.

Results. The results are listed in Table 6.5. As in the other experiments, the performance of the planners was better on the reduced domain. POPF managed to solve all the problems. TFD was able to solve at least the first problem. SGPlan6 solved the first two problems as in the previous experiment, but the time needed for computation was shorter.

Discussion. The results of the tests suggest that in case of interconnected agents, reducing the domain also helps the planners to solve the problems. We also discovered that our assumption that Part 3 of this experiment is more difficult than Part 2 was not completely true. POPF was able to solve the third problem on this domain faster than the second one.

6.5 Interconnected Agents with Time Windows

As in the case of standalone agents, we also performed experiments where valid plans have to include actions which can occur only in particular time windows.

```
(define (problem rpg-problem-exp11-liter)
    (:domain rpg-domain-complete-liter)
    (:objects John James - man Stacy - woman)
    (:init
        ...
    )
    (:goal
        (and
            (take_part_City1_battle_goal John)
            (take_part_City1_battle_goal James)
            (married John)
        )
    )
)
```

Figure 6.8: Experiment 7, Part 1 - extract of the PDDL problem.

```
(define (problem rpg-problem-exp12-liter)
    (:domain rpg-domain-complete-liter)
    (:objects John James - man Stacy Kate - woman)
    (:init
        ...
    )
    (:goal
        (and
            (take_part_City1_battle_goal John)
            (take_part_City1_battle_goal James)
            (married John)
            (married James)
        )
    )
)
```

Figure 6.9: Experiment 7, Part 2 - extract of the PDDL problem.

Experiment 7: Interconnected Soldiers

Hypothesis. If the goals in a PDDL problem require the planner to include actions which can be undertaken only in a specified time window, the complexity of the task for the planner will grow significantly.

Method. We designed PDDL problems analogical to the problems used in the previous two experiments. The requirements on the personal life of the agents are the same as in the previous set of experiments, but this time John and James have to become soldiers and meet in the battle for City1. Extracts of these problems are depicted in Fig. 6.8, 6.9 and 6.10. The tests were performed on the full testing domain. As in the previous case we ran the experiment five times with each planner on both PCs with the time limit of one hour.

Results. The results are depicted in Table 6.6. Introducing actions which can occur only in a certain time window is a big complication for SGPlan6, which needed significantly more time for the first test in the set than in Experiment 5. However, POPF performed better in this experiment than in Experiment 5, solving two of the set of problems, although only on the testing PC with more memory.

```
(define (problem rpg-problem-exp13-liter)
    (:domain rpg-domain-complete-liter)
    (:objects John James - man Stacy Kate - woman)
    (:init
        ...
    )
    (:goal
        (and
            (take_part_City1_battle_goal John)
            (take_part_City1_battle_goal James)
            (married John)
            (married James)
            (had_affair John Kate)
        )
    )
)
```

Figure 6.10: Experiment 7, Part 3 - extract of the PDDL problem.

	SGPlan6				TFD				POPF			
	PC1		PC2		PC1		PC2		PC1		PC2	
	M	SD	M	SD	M	SD	M	SD	M	SD	M	SD
Part 1	1097.97	0.88	1552.41	2.89	0.52	0.02	0.63	0.00	0.22	0.00	0.25	0.00
Part 2	-	-	-	-	-	-	-	-	-	-	105.96	0.31
Part 3	-	-	-	-	-	-	-	-	-	-	-	-

Table 6.6: Experiment 7: Interconnected agents with time windows. Times of computation in seconds (M - mean, SD - standard deviation).

	SGPlan6				TFD				POPF			
	PC1		PC2		PC1		PC2		PC1		PC2	
	M	SD	M	SD	M	SD	M	SD	M	SD	M	SD
Part 1	279.64	1.29	475.58	1.37	68.82	0.33	88.50	0.10	0.12	0.01	0.11	0.00
Part 2	-	-	-	-	797.15	1.18	1143.34	3.33	-	-	-	-
Part 3	-	-	-	-	-	-	-	-	-	-	-	-

Table 6.7: Experiment 8: Interconnected agents with time windows on a smaller domain. Times of computation in seconds (M - mean, SD - standard deviation).

Discussion. The results correspond to our hypothesis. The planners in this tests performed worse than in tests without time windows (Experiment 5) or similarly, although the problems in Experiment 5 needed plans with many more actions to be solved.

Experiment 8: Interconnected Soldiers on a Smaller Domain

To complete our tests of effects of reducing the domain, we repeated also Experiment 7 on a smaller domain.

Hypothesis. As in the previous experiments, reducing the domain will significantly decrease the difficulty of the tasks for the planners.

Method. We repeated the tests from the previous experiment on a smaller domain containing only packages *basic*, *soldier* and *relationships*. The task in part 1 contains 577 grounded predicates, in parts 2 and 3 there are 806 grounded predicates. We again repeated the tests five times with each planner on both PCs.

Results. The results are listed in Table 6.7. SGPlan6 needed less time for computation than in the last experiment, which was anticipated. However, the results of TFD and POPF are surprising. POPF only managed to compute plan for the first problem in this set of tests, although in Experiment 7 it also computed a plan for the second problem. Moreover, although the reduction of the domain enabled TFD to compute a plan for the second problem, the time of computation needed for the first problem was significantly longer than in Experiment 7. Besides, the plan outputted as a solution of the second problem is valid, but surely would not be accepted by a designer, as it contains a lot of marriages and immediate divorces, which are not needed at all.

Discussion. The results show that when the agents are interconnected and we need to include into the plan some actions which can occur only in a specified time windows, we can encounter a very strange behavior of the planners, probably due to the properties of used heuristics. This experiment proves that although the hypothesis that reducing a domain reduces the time of computation is generally true, there are also exceptions from this rule.

6.6 Summary

In this chapter, we have presented our testing domain and the results of a set of experiments we performed. By this, we fulfilled the remaining part of goal 2 and goal 3 of this work, as they are listed at the end of Chapter 1.

The experiments show that the approach we chose works in general. However, using the actual general-purpose planners we were able to solve only simple planning problems, mainly because of insufficient amount of memory. Real problems generated from a designer's requirements would be supposedly much more complex. A designer would probably also create a more comprehensive virtual world, ours is just a prototype for testing with less than 100 actions.

The main performance issue appears when we need to include in the resulting plan some actions which can take place only in certain time windows. The amount of time for computation needed by the planners increases rapidly, or the planners are not able to generate a plan at all. Generation of history of interconnected agents is also more complicated for the planners than generation of history of standalone agents.

A promising possible solution of these issues would be implementing a special-purpose planner equipped with suitable heuristics. This is discussed in the next chapter.

The experiments also demonstrate that an authoring tool developed for the designers should support package management. When we repeated the experiments on a smaller domain, the performance improvement was usually significant. Package management would also facilitate better organization of the actions and states possible in a designer's virtual world, as well as reusability of its parts.

Chapter 7

Future Works

7.1 Special-Purpose Planner

As shown in the previous chapter, actual general-purpose planners have difficulties with our domains and problems, although some of them perform very well on domains used for competitions like the International Planning Competition (IPC), which is held each two or three years during the International Conference on Planning and Scheduling (ICAPS). This issue probably could be solved by implementing a special-purpose planner with heuristics suited for our type of problems.

On one hand, in comparison with IPC domains and problems, our domain contains more operators and predicates and also requires more PDDL features than the majority of IPC domains. For example, from the domains used in temporal satisficing track in IPC-2008 [22], the domain containing the largest number of actions is parcprinter-strips. It contains 13 predicates and 40 durative actions.

On the other hand, our problems are not so complicated. Usually, just a few actions are needed to achieve a particular goal, the resources are not very limited and we do not need to optimize for time. That brings us to the belief that a carefully designed special-purpose planner should improve the efficiency of the computational step of the method a lot and so implementing such a planner is one of the meaningful future works.

7.2 SAT or CSP

The issue with the computation could be also solved by switching from planning to other approaches, like SAT or CSP. Converting a designer's requirements directly into a SAT or CSP task is not very straightforward. However, if we found a way to do it automatically, we could then use some of the almost professional, but freeware SAT resp. CSP solvers, e.g. [15] resp. [19]. Trying this approach is probably the most interesting way in which our research could continue now.

7.3 Authoring Tool

The proposed design method will only be complete with a software tool used to carry out Steps 1, 2, 4 and 5 from the workflow. This tool should enable a designer not only to specify her requirements in a user-friendly way, but should also help her

with managing her domains, creating logical packages (as "mage", "soldier", "relationships"). Above all, it would perform the conversion from the designer's requirements to PDDL, described in Chapter 5, and run the planner. After converting the resulting plan into an intelligible representation (maybe graphical), it would allow the designer to administer her changes to the resulting contents of episodic memory, guiding her through the process and warning her when she is about to perform a change that breaks up some logical constraints.

Although implementing such a tool is an essential step for practical use of our method, at the moment we see solving the issues with Step 3 of the workflow to be the most urgent task to tackle.

7.4 More Detailed World

There are several directions in which we could go to have the resulting contents of episodic memory more detailed. For example, the virtual world used actually for our tests is just a list of places. Later, it could be useful to create a topology of the world. This should be possible conceptually, however, we have not tackled it yet.

The content of episodic memory generated by the method also contains just brief information about the actions which happened – the name of the action, when it took place and which people, locations etc. were involved. For example, John from Experiment 5 knows that he married Stacy in his village on the 201st day of the history, but nothing more. The designer may want to equip her NPCs with more details about the actions, e.g. a detailed description of the mentioned wedding ceremony. In this case, she has to use some suitable mechanisms to accomplish this in the Step 5 of the workflow. One possible systematical approach to this problem is to generate details of memories using the hierarchical approach described in [5].

Chapter 8

Conclusion

We proposed a complex method for automatic generation of episodic memory for virtual agents, intended to be used for instance by game designers. We used planning to perform the crucial step of the method, the computation of coherent content of episodic memory corresponding to a designer's requirements. When implemented completely, the method will enable the designer to specify her requirements on the history of the agent being created. It will also ensure that generated content of episodic memory will comply with predefined logical constraints of the events. Besides, some level of randomness will be included, to create "more animated" agents and also to enable creation of more agents from just one setting. After the automatic generation of memories of the agents, the designer will be enabled to perform her adjustments to create the final version of these memories.

We tested the performance of planning in our task. It works well in simple cases, however, more complicated settings seem to be out of the possibilities of state of the art general-purpose planners. Thus we foresee a necessity to implement a special-purpose planner optimized for this type of problems or to try a different approach to the computational step of the method, like CSP or SAT.

Despite of the actual technical issues with the computational part of the method, we think that this approach, when refined, can facilitate fast enough automatic generation of widely parametrized history of virtual agents.

Bibliography

[1] Bickmore, T., Schulman, D., Yin, L. (2009): Engagement vs. Deceit: Virtual Humans with Human Autobiographies. In *Proc. of Intelligent Virtual Agents '09*, LNCS 5773, 6-19. Springer.

[2] Bioware (2008): Mass Effect. http://masseffect.bioware.com/me1/.

[3] Blizzard Entertainment (2004): World of Warcraft. http://us.blizzard.com/en-us/games/wow/.

[4] Brom, C., Lukavský, J. (2009): Towards Virtual Characters with a Full Episodic Memory II: The Episodic Memory Strikes Back. In *Proc. Empathic Agents* 1-9. AAMAS workshop.

[5] Brom, C., Pešková, K., Lukavský, J. (2007): Modelling human-like RPG agents with a full episodic memory. Technical Report No. 2007/4 of the Department of Software and Computer Science Education, Charles University in Prague.

[6] Burkert, O. (2009): Connectionist Model of Episodic Memory for Virtual Humans. Master thesis, Charles University in Prague.

[7] Castellano, G., Aylett, R., Dautenhahn, K., Paiva, A., McOwan, P. W., Ho, S. (2008): Long-term affect sensitive and socially interactive companions. In *4th Int. Workshop on Human-Computer Conversation.*

[8] Coles, A. J., Coles, A. I., Fox, M., Long, D. (2010): Forward-Chaining Partial-Order Planning. In *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS-10).*

[9] Cresswell, S., Coddington, A. (2003): Planning with timed literals and deadline. In *UK Planning and Scheduling SIG (PlanSig)* 22-35.

[10] Dias, J. (2005): Fearnot!: Creating emotional autonomous synthetic characters for empathic interactions. Master's thesis, Universidade Técnica Lisboa.

[11] Dias, J., Ho, W.C., Vogt, T., Beeckman N., Paiva, A., Andre, E. (2007): I Know What I Did Last Summer: Autobiographic Memory in Synthetic Characters. In *Proc. of ACII* 606-617. Springer-Verlag.

[12] Doherty, D., O'Riordan, C. (2008): Toward More Humanlike NPCs for First-/Third-Person Shooter Games. In *AI Game Programming Wisdom IV* 499-512. Charles River Media.

[13] Ebert, D. S., Musgrave, F. K., Peachy, D., Perlin, K., Worley, S. (2003): Texturing & Modelling - A Procedural Approach. Morgan Kaufmann.

[14] Edelkamp, S., Hoffmann, J. (2004): PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition. Technical Report 195, Albert-Ludwigs-Universität Freiburg, Institut für Informatik.

[15] Eén, N., Sörensson, N. (2005): Minisat a sat solver with conflict-clause minimization. In *SAT 2005 Competition*.

[16] Eyerich, P., Mattmüller, R., Röger, G. (2009): Using the Context-enhanced Additive Heuristic for Temporal and Numeric Planning. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009)*.

[17] Fox, M., Long, D. (2003): PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. In *Journal of Artificial Intelligence Research* **20**, 61-124.

[18] Funcom (2008): Age of Conan: Hyborian Adventures. http://www.ageofconan.com.

[19] Gecode: a Generic Constraint Development Environment. http://www.gecode.org

[20] Glassner, A. (2004): Interactive Storytelling: Techniques for 21st Century Fiction. A. K. Peters, Natick.

[21] Greuter S., Parker J., Stewart N., Leach G. (2003): Real-time procedural generation of 'pseudo infinite' cities. In *Proceedings of GRAPHITE 2003* 87-95. ACM Press.

[22] Helmert, M., Do, M., Refanidis, I. (2008): International planning competition ipc-2008, the deterministic part. http://ipc.informatik.uni-freiburg.de/.

[23] Helmert, M., Geffner, H. (2008): Unifying the causal graph and additive heuristics. In *Proceedings of ICAPS 2008*, 140-147.

[24] Hirst, W., Manier, D. (1995): Remembering as communication: A family recounts its past. In *Remembering Our Past: Studies in Autobiographical Memory* 271-290. Cambridge University Press.

[25] Ho, W., Dautenhahn, K., Nehaniv, C. (2005): Autobiographic Agents in Dynamic Virtual Environments - Performance Comparision for Different Memory Control Architectures. In *Proc. IEEE CEC*, 573-580.

[26] Ho, W. C., Watson, S. (2006): Autobiographic knowledge for believable virtual characters. In *Proc. of Intelligent Virtual Agents*, LNCS 4133, 383-394. Springer-Verlag.

[27] Hoffmann, J. (2003): The Metric-FF planning system: Translating "Ignoring Delete Lists" to numeric state variables. In *Journal of Artificial Intelligence Research* **20**, 291-341.

[28] Hsu, C.-W., Wah, B. W. (2008): The SGPlan planning system in IPC6. In *6th International Planning Competition Booklet* (ICAPS-08).

[29] Kučerová, L., Brom, C., Kadlec, R. (2010): Towards Planning the History of a Virtual Agent. In *Proceedings of ICAPS'10 Workshop on Planning in Games.*

[30] Li, B., Riedl, M. O. (2010): Planning for Individualized Experiences with Quest-Centric Game Adaptation. In *Proceedings of ICAPS'10 Workshop on Planning in Games.*

[31] Loyall, B. A. (1997): Believable Agents: Building Interactive Personalities. Ph.D. dissertation. Carnegie Mellon University.

[32] McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., Wilkins, D. (1998): PDDL – The Planning Domain Definition Language – Version 1.2, Techical Report, CVC TR-98-003, Yale Center for Computational Vision and Control.

[33] Meehan, J. R. (1976): Tale-spin, an interactive program that writes stories. In *Proceedings of the fifth international joint conference on artificial intelligence*, 91-98. Cambridge, Massachussets.

[34] Nuxoll, A. (2007): Enhancing Intelligent Agents with Episodic Memory. Ph.D. thesis, The University of Michigan.

[35] Pérez y Pérez, R., Sharples, M. (2001): Mexica: A computer model of a cognitive account of creative writing. *Journal of Experimental and Theoretical Artificial Intelligence* **13**, 119-139.

[36] Porteous, J., Cavazza, M. (2009): Controlling Narrative Generation with Planning Trajectories: The Role of Constraints. In *Proc. Of 2nd Int. Conf. on Interactive Digital Storytelling.* LNCS 5915, 280-291, Springer.

[37] Prusinkiewicz, P., Lindenmayer, A. (1990): The Algorithmic Beauty of Plants. Springer-Verlag, New York.

[38] Rickel, J., Johnson, W. L. (1999): Animated Agents for Procedural Training in Virtual Reality: Perception, Cognition, and Motor Control. App. *Artificial Intelligence* **13**(4-5), 343-382.

[39] Riedl, M. (2009): Incorporating authorial intent into generative narrative systems. In *Intelligent Narrative Technologies II*, TR SS–09–06, pp. 91–94. AAAI, Menlo Park.

[40] Riedl, M. O., Young, R. M. (2006): Story Planning as Exploratory Creativity: Techniques for Expanding the Narrative Search Space. New Generation Computing 24(3): 303-323.

[41] Si, M., Marsella, S.C., Pynadath, D.V. (2005): Thespian: Using multi-agent fitting to craft interactive drama. In *AAMAS*, pp. 21–28. ACM, New York.

[42] Trescak, T., Esteva, M., Rodriguez, I. (2010): A Virtual World Grammar for automatic generation of virtual worlds. In *The Visual Computer* **26**(6-8), 521-531. Springer.

[43] Tsang, E.P.K. (1993): Foundations of Constraint Satisfaction. Academic Press, London and San Diego.

[44] Tulving, E., Donaldson, W. (1972): Organization of Memory. Academic Press, New York.

[45] Williams, H.L., Conway, M.A., Cohen, G. (2008): Autobiographical memory. In *Memory in the real world* 21-90. Psychology Press.

[46] Wooldridge, M. (2002): An Introduction to MultiAgent Systems. John Wiley & Sons.

Appendix A

Attachments

This thesis goes accompanied by a CD containing:

- Source code and/or binaries of all the planners used for the experiments (SGPlan6, TFD and POPF).
- All PDDL domains and problems used for the experiments.
- Results of the experiments.
- Shell scripts for easy replication of the experiments.
- README.TXT – guide for step by step installation of the planners and replication of the experiments.
- Text of this thesis in PDF.

Appendix B

Conversion Algorithms

Name: ConvertParameters
Input: List of parameters *P*
Output: PDDL definition of parameters of a function, predicate or action

foreach parameter *Param = (pid, t)* **in** *P*
    write " ?", *pid*
    if *t* != null write " - ", *t*
writeline

Figure B.1: Algorithm to convert parameters.

Name: ConvertInstantiated
Input: Instantiated state, action or function *I = (id, CP)*
Output: Definition of a part of PDDL condition of a durative action

write "(", *id*
if *I* is instantiated action write "_goal"
foreach identifier *ident* **in** *CP*
    if *ident* is parameter identifier write " ?" else write " "
    write *ident*
write ")"

Figure B.2: Algorithm to convert an instantiated state, an action or a function.

Name: ConvertStates
Input: List of states S, list of actions A
Output: Definition of PDDL predicates belonging to the domain

```
writeline "(:predicates"
foreach action Action = (aid, P, d, s, e, pb, ns, c, E) in A
    if pb != null
        write aid, "_init"
        ConvertParameters(P)
    write aid, "_goal"
    ConvertParameters(P)
    if s != null writeline aid, "_going_on"
foreach state State = (sid, P) in S
    write sid
    ConvertParameters(P)
writeline ")"
```

Figure B.3: Algorithm to convert states.

Name: ConvertActions
Input: List of actions A
Output: Definition of PDDL durative actions belonging to the domain

```
foreach action Action = (aid, P, d, s, e, pb, ns, c, E) in A
    writeline "(:durative-action ", aid
    if P != null
        write ":parameters
        ConvertParameters(P)
    writeline ":duration (= ?duration ", d, ")"
    if c != null or pb != null
        ConvertActionCondition(A)
    writeline ":effect"
    writeline "(and"
    ConvertEffects(E)
    write "(at end (", aid, "_goal"
    foreach parameter Param = (pid, t) in P
        write " ?", pid
    writeline "))"
    writeline ")"
```

Figure B.4: Algorithm to convert actions.

Name: ConvertCondition
Input: Condition *Cond*
Output: Definition of one PDDL condition of a durative action

if *Cond* is atomic condition
    if *Cond* is negated write "(not "
        if *Cond* is instantiated state or instantiated action
            ConvertInstantiated(*Cond*)
            writeline
        elseif *Cond* is function equation
            Cond = (IF, val, oper)
            write "(", *oper*, " "
            ConvertInstantiated(*IF*)
            writeline " ", *val*, ")"
    if *Cond* is negated write ")"
    writeline
elseif *Cond* is boolean condition
    Cond = (oper, C)
    writeline "(", *oper*
    foreach condition *c* in *C* ConvertCondition(*C*)
    writeline ")"

Figure B.5: Algorithm to convert a condition of an action.

Name: ConvertEffects
Input: List of effects *E*
Output: PDDL definition of effects of an action

foreach effect *Effect = (tv, ae)* **in** *E*
    if *tv* == "start" write "(at start (" **else** write "(at end ("
    if *ae* is negated write "not ("
    if *ae* is instantiated state ConvertInstantiated(*ae*)
    if *ae* is function change
        ae = (fid, CP, val)
        if *val* > 0 write "(increase (" else write "(decrease ("
        write *fid*
        foreach identifier *ident* **in** *CP*
            if *ident* is parameter identifier write " ?" else write " "
            write *ident*
        write ") ", *val*, ")"
    if *ae* is negated write ")"
    writeline "))"

Figure B.6: Algorithm to convert effects of an action.

Name: ConvertGoalState
Input: Goal state *GoS = (GS, GA, GFE)*, list of noise actions and person identifiers *NA*, list of actions *A*
Output: A valid PDDL definition of goal state

writeline “(:goal”
foreach grounded state *GState = (sid, CO)* **in** *GS*
    if *GState* is negated write “(not ”
    write “(”, *sid*
    foreach constant or object identifier *id* **in** *CO*
        write “ ”, *id*
    if *GState* is negated write “)”
    writeline “)”
foreach grounded function equation *GFEq = (GF, val, oper)* **in** *GFE*
    GF = (fid, CO)
    write “(”, *oper*, “ (”, *fid*
    foreach constant or object identifier *id* **in** *CO*
        write “ ”, *id*
    writeline “) ”, *val*, “)”
foreach grounded action *GAction = (aid, CO)* **in** *GA*
    write “(”, *aid*, “_goal”
    foreach constant or object identifier *id* **in** *CO*
        write “ ”, *id*
    writeline “)”
foreach *P = (aid, pid)* **in** *NA*
    writeline “(”, *aid*, “_goal ”, *pid*, “)”
writeline “)”

Figure B.7: Algorithm to convert a goal state.