COLLABORATION WITH AGENTS IN VR ENVIRONMENTS

THÈSE N^{O} 3350 (2005)

PRÉSENTÉE À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

Institut des systèmes informatiques et multimédias

SECTION D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Jan CIGER

Magister en informatique, Université Comenius, Bratislava, Slovaquie
et de nationalité slovaque

acceptée sur proposition du jury:

Prof. D. Thalmann, directeur de thèse
Prof. B. Faltings, rapporteur
Prof. A. Nijholt, rapporteur
Prof. P. Petta, rapporteur

Lausanne, EPFL
2005

Acknowledgments

During my four years at LIG/VRlab I had the pleasure to work with many smart people and to learn a lot from them. I wish to thank to all of them for their wisdom and patience while I was working on this dissertation. It is impossible to name everybody I have worked with, but I want to at least express my thanks:

To professor Daniel Thalmann for giving me the opportunity to conduct this research and for the great work conditions and learning opportunity in his laboratory during these four years.

To Josiane Bottareli and Zerrin Celebi for their help with various administrative problems and issues.

To all my colleagues from the laboratory I have worked with, but my special thanks go to Bruno Herbelin, Tolga Abaci and Branislav Ulicny for their collaboration, keen advice, friendship and support during the more difficult times.

To the great EPFL students I have enjoyed to work with. Their help with the implementation of the case studies for this thesis and their patience with my pedagogical "skills" is really appreciated. I wish to thank namely to Karim Krichane for his excellent work on the "Virtual Guide" application, Thibault Genessay for giving a more user-friendly face to my agents with the problem-solving user interface and Fabrice Hong for the "Natural language interface" application.

To Miguel García Arribas, who was less a student and more a fellow researcher, collaborator and a friend. He has also implemented the "City riot" application.

To my family for the support and encouragement to go ahead with my studies.

Finally, I want to thank to Silvia for her love, support and patience during those four long years.

Abstract

Virtual reality is gaining on importance in many fields – scientific simulation, training, therapy and also more and more in entertainment. All these applications require the human user to interact with virtual worlds inhabited by intelligent characters and to solve simulated or real problems.

This thesis will present an integrated approach to simulated problem solving in virtual reality environments, with the emphasis on teamwork and the ability to control the simulations. A simulation framework satisfying these goals will be presented.

A unified approach to the representation of semantic information in virtual environments based on predicate calculus will be introduced, including the representation of the world state, action semantics and basic axioms holding in the simulated world.

Afterwards, the focus will be on the collaboration model based on task delegation and facilitator-centric architecture. A simple but efficient facilitator design will be presented.

The issues of the collaborative problem solving will be examined. A new technique using propositional (STRIPS-like) planning with delegated actions and object-specific planning will be described.

A control technique for virtual characters/objects will be detailed, enabling run-time exchange of control and control sharing over a virtual entity between multiple autonomous agents and/or human users.

Finally, a set of case studies will be shown, illustrating the possible applications of the techniques developed and described in this dissertation.

Version abrégée

La réalité virtuelle prend une importance grandissante dans de nombreux domaines d'application – simulations scientifiques, formation, thérapie et aussi de plus en plus pour les divertissements. Toutes ces applications poussent l'utilisateur à interagir avec un monde virtuel et lui permettent de résoudre des problèmes simulés ou réels.

Cette thèse présente une approche de la résolution de problèmes simulés en environnements virtuels, approche axée sur l'intégration des outils de collaboration en équipe et de contrôle de la simulation. Une plateforme de développements satisfaisant ces objectifs y sera présentée.

Nous présenterons une méthode de représentation de données sémantiques dans les environnements virtuels basée sur la logique des prédicats permettant la description de l'état du monde virtuel et de la sémantique des actions ainsi que la définition d'axiomes de base valides dans ce monde simulé.

Puis, notre attention se portera sur le modèle de collaboration basé sur la délégation de tâches avec une architecture centrée sur un facilitateur. Un modèle simple mais efficace d'un tel facilitateur sera présenté.

Nous examinerons ensuite la résolution collaborative de problèmes et présenterons une nouvelle technique utilisant la planification propositionnelle avec délégation des actions et planification par objets.

Une technique de contrôle des personnages/objets virtuels sera détaillée afin de préciser comment il est possible d'échanger le mode de contrôle pendant l'exécution et comment plusieurs agents autonomes et des utilisateurs humains peuvent se partager le contrôle d'une entité.

Enfin, nous présenterons plusieurs cas d'étude illustrant les applications possibles des techniques développées et décrites dans ce mémoire.

Contents

List of Figures

Chapter 1

Introduction

I'm sorry, Dave, I'm afraid I can't do that...

HAL 9000 in "2001: A Space Odyssey", 1968

The technology is advancing in huge steps and computers are getting more and more powerful every day. However, one fundamental challenge remains – how do the humans communicate and collaborate with the machine? More often than not the process could be described by the quote from the famous movie by Kubrick and Clarke – the user asking the machine to work and the computer's failure to do so, accompanied by an incomprehensible error message. HAL was at least communicating in a human language, the ordinary computers are still not.

The man–machine interaction is a vast topic being worked on by many. The evolution of user interfaces took us from wires and pins on machines like ENIAC[1] to the WIMP paradigm[2] and virtual reality of today. Interaction is only one side of the story – the user wants the machine to work for him. The collaboration between these two entities, bridging the gap between the "meatspace"[3] and the virtual world on the other side will be the main topic of this document.

1.1 Motivation

With the advances in the technology, virtual reality simulations are gaining importance. What was completely unfeasible only few years ago is possible today and will be probably in routine use few years in the future. Virtual reality is often deployed to simulate situations which are either dangerous, expensive, unpredictable or simply impossible to perform in the real world. Typical examples are military training, training of machinery operators, emergency situation simulations and many others. All these applications frequently require the user to correctly interact with the virtual world and to solve simulated or real problems.

Unfortunately, most virtual reality systems are built as single-purpose applications, without the flexibility or capabilities required for problem solving. It is rarely possible to "let the user loose" in the system, have him/her work at own tempo and find their own way to solve a given problem if the system knows how to respond only in few fixed ways to few pre-defined challenges.

[1]Where the programs were literally "hardwired".

[2]WIMP – windows, icons, menus, pointing device.

[3]meatspace: <jargon> The physical world (as opposed virtual reality) where you might spend facetime with the carbon community, 15.01.1999, The Free On-line Dictionary of Computing, ©1993-2004 Denis Howe.

Another aspect is extensibility of the simulations – except in trivial cases, it is almost never possible to insert a new object into the application and expect the rest of the system to seamlessly work with it, without changes. Such extensions usually require complex modifications in many parts of the simulator which need to be made aware of the change.

Both problems – the inflexibility and problematic extensibility of the virtual reality applications are caused by a general lack of high-level semantic information – metadata. This also makes it very difficult to apply known problem solving techniques to the virtual worlds, e.g. planning, state space search, constraint solvers etc. which can only work with high level information.

Virtual reality applications often try to simulate the real world and populate the virtual environment with simulated characters (e.g. virtual humans) with various degrees of intelligence. Such simulations often require the user to interact with the virtual characters present in the scene but unless some communication and collaboration formalism ("common language") is in place, such interactions can be only very limited in scope.

1.2 Objectives of the research

With the focus on the virtual reality field, there are several main objectives of this work which will be explored in the subsequent chapters.

First, the work introduces unified representation of the semantic information in the virtual world based on propositional logic. This approach is known from the artificial intelligence, however it was usually applied only to the active parts of the simulation (e.g. intelligent agents) as a part of their behavior system. This work will introduce an extension of the smart object technique originally developed by Kallmann [51] towards high-level reasoning where the objects in the virtual world are completely self-contained – they include geometric information, basic semantic information for animation purposes and the newly added high level semantic information (metadata) required for the interaction and reasoning about them. Such extension enables the seamless extensibility of the virtual environments because the active parts of the simulation can learn the required information at run time directly from the objects.

The second objective is to introduce a method for human–intelligent agent collaboration enabling the flexibility required by the typical virtual reality applications. This objective consists of several sub-goals:

- Different modes of control. Enabling the user and application designer to use different modes of control (e.g. first-person direct interaction, direct orders to the virtual characters, indirect control by proxy, such as an agent directing a team working on the user's behalf). Letting the user to select the preferred mode allows for individual problem-solving styles. It also provides an important fallback mechanism for cases when the autonomous characters do not behave in the desired way – direct intervention of the user can help solve a problem which may be unsurmountable otherwise. This thesis will present a technique allowing for such mode change at run-time of the simulation.

- Task delegation. It is often beneficial to let the computer do the work for user. The task could be either too boring or too difficult to be performed by a human. In either case, enabling the user to offload part of his workload to the simulation system allows him to focus on his core activity – e.g. a main training objective while letting the machine handle the rest of the work, for example already accomplished training phases. A facilitator-based technique will be presented that enables the user to delegate tasks to the intelligent agents for solution.

- Automatic sub-task solving. To achieve a realistic behavior it is not enough to let the user to only delegate the task to the simulation system but the simulator has to be able to solve it autonomously in order to be really useful. This goal can be achieved by employing known artificial intelligence techniques, such as planning. Two extensions to the common planning techniques based on the Graphplan algorithm will be presented, specifically planning with delegation and object-specific planning.
- Teamwork. Work in a team is a typical part of the every day's life and it is often desirable to introduce similar capability to the virtual reality systems. Many tasks are difficult to solve without collaboration of some kind and in some cases the collaboration and coordination itself is a goal – typical example is tactical training for military or police units. A simple teamwork model fitting the most common scenarios will be examined and described.

The final objective of this thesis is to demonstrate how a flexible and reusable framework for human–intelligent agent and agent–agent collaboration in virtual environments can be built, end-to-end from the low level animation issues to the high level artificial intelligence framework. The existing approaches focus either on the low level graphics and interaction issues (typically virtual reality research) or only on the abstract high level artificial intelligence problems, leaving a large gap between the two which makes it difficult to apply the results of the both fields in the same application.

1.3 Contribution

The main contribution of this thesis is a new approach to the human–intelligent agent collaboration and problem solving suitable for use in the virtual environments. In the course of this work several new solutions to the related topics were found and will be covered in this document:

- Method enabling change of mode of control, allowing the user to share control of the virtual characters with the intelligent agents.
- Knowledge and semantic data representation in the virtual environment, based on the propositional logic, with the intent to enable high-level reasoning about the data by intelligent agents.
- Collaboration architecture based on task delegation, facilitation and planning with the focus on problem solving and teamwork for both human – intelligent agent and agent – agent cases.
- Several enhancements to the known planning approaches, specifically integrating the delegated actions and object-specific planning into the standard planners.

1.4 Plan of the thesis

This document is organized into several chapters:

- In the chapter 2, related work relevant to this research will be reviewed.
- Chapter 3 will be dedicated to the symbolic representation of the virtual world, required for any subsequent high-level manipulation.
- Chapter 4 will examine the proposed collaboration model for the collaboration between both humans and intelligent agents and among the agents themselves.

- In the chapter 5, the document will focus on the aspects of collaborative problem solving in the virtual environments, with the emphasis on the issues of planning.
- The chapter 6 will be dedicated to the implementation of the proposed approach and demonstration of the functionality of the individual components.
- Chapter 7 describes six case studies which were used to validate the results of this work.
- Finally, in the chapter 8, the contributions of the presented work will be summarized and the directions for the further research outlined.

Chapter 2

Related work

This chapter makes an overview of the terminology, concepts and related work relevant to the topic of this thesis. It is divided into several sections roughly corresponding to the parts of the thesis. In the first part the background information on semantic information in the virtual environments is presented, introducing concepts such as smart objects. The next part provides basic information on agents and multi-agent systems. Afterwards, an overview of the collaboration techniques and their accompanying theories is presented, together with the overview of the related technologies and applications known from the published works. The final part will focus on planning, planners and their applications in the virtual reality simulations.

It is assumed that the reader is familiar at least to some extent with the discussed topics and therefore the terms and topics are not discussed in exhaustive manner.

2.1 Definitions of the frequently used terms

- *Agent.* There is a no clear definition for the term *agent.* According to Russell and Norvig [91], "an agent is anything that can be viewed as perceiving its **environment** through **sensors** and acting upon that environment through **actuators**."

  Another definition can be found in Jennings [49]: "An agent is an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives."

  The second definition is more general, because not all agents have to change their environment or have sensing capabilities. However, the important points are *flexibility* and *autonomous* action.

- *Intelligent agent.* In the virtual reality simulation context, agents are often described as intelligent to emphasize the AI functionality. They are frequently embodied as virtual characters.

- *Virtual human.* A humanoid character in the virtual environment, frequently controlled either by an *agent* or representing the user in the virtual world. In the latter case it is called an *avatar*.

2.2 Semantic information in VR environments

Virtual reality applications often require the virtual characters to be able to manipulate the objects in their environment. Such interactions can be arbitrarily complex. Traditional solutions are predesigned or pre-recorded (e.g. by motion capture) animations. Another, more general solution is to shift the responsibility for the animation at least partially to the object, leading to the smart object concept [51].

The idea of smart objects is based on augmenting the basic geometric information such as polygonal mesh by a set of semantic data enabling the virtual character to properly manipulate the object. Such semantic data can be for example hand positions for grasping, various important positions on the surface of the object (e.g. denoting locations of buttons operating model of a machinery) or symbolic information describing the general properties of the object – e.g. its weight, color, shape etc. The virtual character – smart object interaction is usually handled by the sets of predefined animations, most often in a scripted form.

There is a number of works in the literature that have addressed similar issues. The Improv system described by Perlin and Goldberg [80] consists of an Animation Engine, used for the motion generation aspects and a Behavior Engine, used for describing the decision-making process through rules. The Behavior Engine allows scripting in a language close to normal English, making it accessible for the non-programmers as well.

Parametrized action representation [8] by Badler et al. describes an action by specifying information about the pre- and post-conditions of an action, its execution steps and which objects are concerned by it. The actions can prescribe chaining of the actions allowing for complex behaviors. The actions come in two forms – uninstantiated (UPAR) and instantiated (IPAR). The uninstantiated actions do not specify the virtual character nor the objects involved in the action, essentially representing whole class of possible actions. Instantiated actions are essentially UPARs extended with the references to the virtual character, objects involved and termination conditions. Collection of UPARs represents all the actions possible in the system.

On the animation front, virtual human – object interaction techniques were specifically addressed in the object specific reasoner (OSR) [62, 63]. The primary aim of this work is to bridge the gap between high-level AI planners and the low-level actions for objects, based on the observation that objects can be categorized with respect to how they are to be manipulated. OSR differs from the work presented in this document in that Levison uses a "top-down" approach. Generic actions are gradually refined by the agent using object taxonomies into executable actions. The agent classifies the object into a category (e.g. a crate belongs to the "containers" category) and then uses that information to decide what to do with it. (e.g. containers can be opened by hands).

More recently, Vosinakis and Panayiotopoulos have introduced the Task Definition Language [110], aimed at filling the gap between higher-level decision processes and an agent's interaction with the environment. This language supports complex high-level task descriptions through combination of parallel, sequential and conditionally executed sequences of primitive actions. The primitive actions can be of two types – either *predefined* with fixed duration or *goal oriented*, where the termination condition is given for the action.

The smart objects paradigm has been introduced for interactions of virtual humans with virtual objects by Kallmann [51]. With smart objects, all the interaction features of an object are contained in the object specification itself. Apart from the properties of the object itself (e.g. animation description, geometry etc.), they include semantic information aiding the interacting agents to perform the animation – such as expected hand positions, position and the orientation of the virtual human relative to the object. Another distinct feature of smart objects is the availability of *interaction plans* – scripts defining the animation sequence for a particular interaction – e.g. a virtual human opening a drawer or operating an elevator (see fig. 2.1).

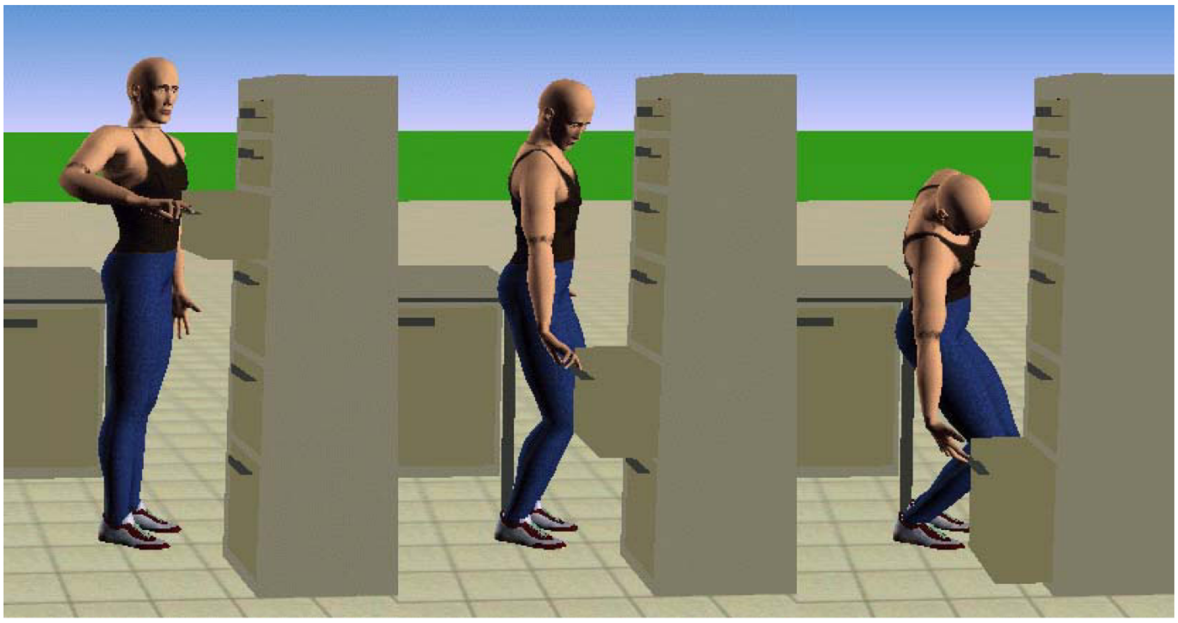


Figure 2.1: Virtual human operating a smart object – a drawer

Smart objects as introduced by Kallmann have a very important property – they allow easy reusability of already defined objects. This property stems from the fact that the smart objects are self-contained and the animation control is decentralized, alleviating the need for complex update of the agents in the simulation whenever a new object is added to the virtual environment. Kallmann's work was continued and extended by Abacıtowards automatic motion generation, such as automated grasping, inverse kinematics or motion planning – see [4].

Unfortunately, the smart objects, as introduced by Kallmann and Abacı, are targeted strictly towards the animation – the semantic information contained within them is animation-oriented and geometric in nature. There is no explicit support for higher-level reasoning and symbolic description of the properties of the objects in their implementation.

Work by Farenc [24] presents a technique for adding semantic information into the simulated urban environment. The author calls the resulting virtual city an "informed environment". The information is organized in hierarchical manner by space partitioning. For example the block of buildings is subdivided into buildings and streets. Streets are further subdivided into sidewalks and roads. Roads are divided by junctions and into segments. Each segment in turn contains information about objects contained in it, such as crosswalks, bus stops, signs, benches etc.

The hierarchically organized information is then used by intelligent agents (pedestrians) navigating in the simulated city in several ways. The inherent space partitioning enables fast queries about the objects and properties of the agent's surroundings. Another use of the information is to make informed decisions by the agents, such as that it is allowed to walk only on sidewalks and crosswalks, but not in the middle of the road. Embedding the information directly into the environment allows for large decentralization of the simulation and simpler agents. However the preparation of the fully augmented geometric model is very time consuming and difficult due to the sheer amount of the data – for example typical model of a city quarter as used by Farenc can contain several thousands of primitives of many types (such as polygons modeling sidewalk pieces, benches, trees, bus stops, etc.).

2.3 Agent-based systems

This section will focus on the notion of agency, agents, multi-agent systems and their role in the virtual reality simulations.

As defined before, an *agent* is an autonomous entity exhibiting qualities of *agenthood* – perceiving its environment and acting on it through actuators. In the context of this work most agents will be considered as standalone software entities, *intelligent* and *embodied*, represented in the virtual world in the form of a virtual character (having an avatar).

However, several important agents encountered in this thesis will be non-embodied, serving in auxiliary roles – such as various planners, brokers or user interface agents. These do not have direct visual representation in the virtual environment but they provide important services to the rest of the system.

2.3.1 Multi-agent systems

According to Luger [64], the field of distributed artificial intelligence (DAI) has its origins at MIT around 1980. A group of researchers around Rodney Brooks worked on the concept of multiple problem solvers working together to solve a single task. Their goal was an attempt to explore how different problem solvers could be coordinated together while working on a single task.

The distributed problem solving does not require centralized store for knowledge[1]. Such decentralization allows for specialization of the problem solving components, where each of them focuses on solving a single particular aspect of the problem.

Second aspect of DAI is that the solvers are *situated* in their environment, enabling them to offload parts of the problem solving process into their environment. The individual solver does not need to know the progress toward the final goal.

Luger defines multi-agent system as a "computer program with problem solvers situated in interactive environments, which are each capable of flexible autonomous, yet socially organized action that can, but need not be, directed towards predetermined objectives or goals."

The key properties of a multi-agent system are:

- *Situatedness*. Each solver (agent) works in a defined environment, from which it can receive input and can change it using actuators.
- *Autonomy*. Each agent works on its own and independently from the rest of the system. It has control over its own actions and its internal state. This property distinguishes agents from e.g. objects in the object oriented programming domain – objects are rarely autonomous. Agents have typically their own threads of control and exist standalone, the same is usually not true about objects.
- *Flexibility*. Each agent is both *reactive* and *proactive*, that means that it not only reacts to the inputs it receives from the environment but also can be goal driven and opportunistic, responding to the changing environment by creating alternatives for its behavior. For example an agent controlling a hostile video-game character can anticipate the moves of the player before they actually happen and adapt its behavior accordingly.
- *Social behavior*. Agents are social because they can interact with other agents, software components or humans. The agent is only a part of a larger system which is solving a given problem.

[1]Even though it can be beneficial to have it as will be shown in the later chapters of this document.

These four properties of a multi-agent system bring also many challenges and drawbacks. For example, how can the agents communicate among themselves and coordinate their activities? How should they perceive the environment? Because they are situated and the knowledge about the environment is decentralized, they have their own *beliefs* about the state of the world which may be inaccurate and/or outright wrong. These problems were addressed by the researchers in the past and some of these works will be reviewed in the next section.

2.4 Collaboration

According to a dictionary *to collaborate* means “to work together, especially in a joint intellectual effort.” Collaboration is a key activity in multi-agent systems where multiple agents work together on finding a solution to the problem. However, collaboration may be also between an agent(s) and a human user where the machine helps the human to solve a specified task.

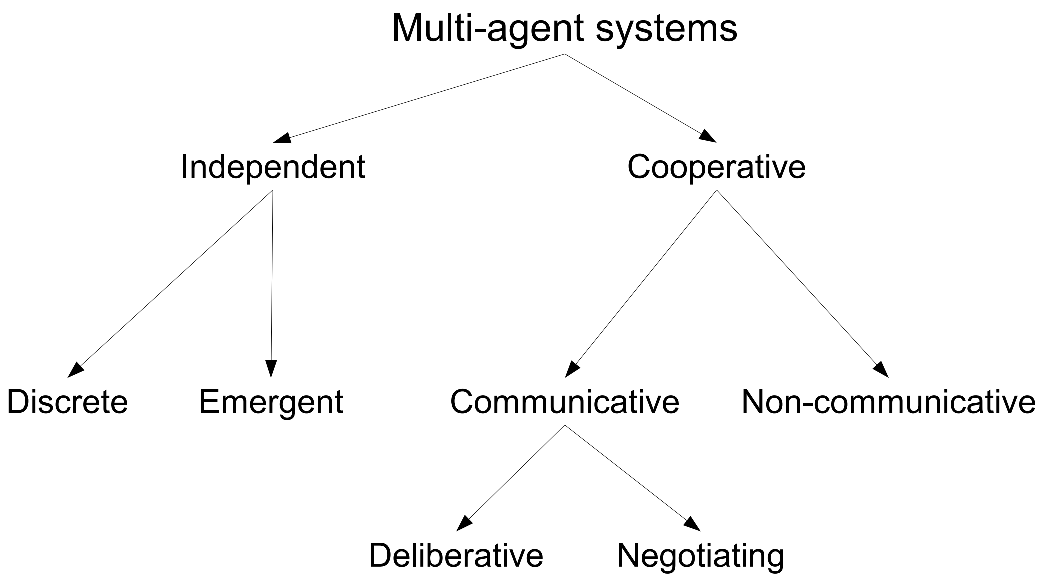


Figure 2.2: Types of collaborative multi-agent systems

Doran et al. in [18] classifies the multi-agent systems into several categories depending on the type of collaboration, as seen in figure 2.2. The main division is into independent and cooperative systems. The independent systems can be separated into two categories:

- *Discrete* – agents in such a system do not collaborate at all.

- *Emergent* – the collaboration is not conscious effort by the agents which only try to follow their own goals. The agents appear to cooperate, however from the agent's viewpoint they do not.

The cooperative systems can be divided into communicative and non-communicative. Communicative agents communicate by intentional sending of messages to each other, non-communicative cooperate by observing each other and reacting to each others' behavior.

The communicative systems can be further subdivided into:

- *Deliberative* – deliberative agents try create a common plan of action, that may or may not involve coordination.

- *Negotiating* – negotiating agents are similar to deliberative ones, however there is an aspect of competition (e.g. the agents compete among themselves to get first to the ball in the simulated Robocup competition and then establish a plan how to score a goal.)

The focus of this work will be on deliberative agents, which plan and coordinate together their actions.

There are several published types of applications of multi-agent systems with focus on collaboration, either between the agents and the human users or among the agents themselves. Furthermore, the research on collaborating agents led to the development of multiple collaboration theories and important platforms which will be reviewed in the following subsections.

2.4.1 Purpose

Multi-agent collaborative systems serve different purposes. They are most often found in various military simulations (e.g. STEVE or STEAM-based helicopter pilots), emergency response training (works by M. Tambe), robotics (e.g. RoboCup competition and its simulated variant), team collaboration training and many others.

The work of Rickel and Johnson are of particular relevance. In [88, 89, 90] intelligent agent system STEVE is described. It is intended for task demonstration and training purposes in various domains. The system is capable of spoken communication with the student and demonstrating various activities based on a predefined script of a lesson. The system adapts to the users skills and the student is able to advance at his/her own speed. For example, it is possible to skip parts of the training the user is already familiar with. Alternatively, the student can request repeating and new demonstration of an activity together with an explanation why and how the activity should be performed. Originally the system supported only single agent tutoring a single student, however the later versions support also team work training with multiple STEVE agents and multiple human participants in the virtual environment where the team of human trainees has to work together with their virtual coaches to troubleshoot a simulated problem.

From the technical point of view, STEVE is an immersive application utilizing HMD[2] and data gloves for the human operators. In addition, the application deploys one or multiple instances of the STEVE agent. All communication is conducted verbally, using speech recognition and synthesis equipment. The agents consist of three main modules – perception, cognition and motor control. The perception module tracks the activities of the human users and other agents and provides this information into the cognition module. The cognition module is implemented in Soar [59], which will be discussed in more detail later. The cognition module drives the agent's reactions to the actions

[2]Head Mounted Display

of the users and other agents and generates motor control actions which are then performed by the agent's avatar.

The knowledge in the STEVE system is represented in the form of predefined tasks connected together by causal links. Each task is defined as a plan (i.e. a sequence of either primitive or composite actions) and the causal links define that the task is a pre-requisite for another task. The agent then builds and maintains a partially ordered worst case plan on how to successfully accomplish the task from the current state of the virtual world by keeping track of which pre-requisites were accomplished already and which are still not. Such planning enables it to react to the unexpected actions of the human trainees without having to script all possible contingencies.

Another similar system was described by Evers and Nijholt in [23]. The Jacob system demonstrates a collaborative system where the autonomous agent is used to demonstrate and tutor the human user on performing a simple task – solving the "Towers of Hanoi" puzzle. The focus of the project was to explore the possibilities of using a multi-modal agent-based system in virtual reality environment.

The idea of enabling multimodal communication and interaction in order to offer the user multiple ways of achieving the goal and to adapt to user's preferred way of working was voiced in [77] by Nijholt and Hulstijn. The work is focused on the communication and interaction aspects between (possibly collaborating) agents and human users in virtual reality scenarios.

Many of the works by Milind Tambe belong into another category of collaborative systems. He focuses on the theoretical aspects of the collaboration process, with many applications in different domains, such as military helicopter pilots (the "helo" domain) mentioned in [73] or emergency response training and coordination described in [92, 104] and RoboCup [103]. The results of this group include creation of the frameworks like STEAM [101] and TEAMCORE [86, 105, 104].

Works by Tambe and his team try to perform a quantitative analysis of the teamwork, team forming, allocation of the roles and collaboration inside of the heterogeneous teams combining software agents, humans and robots. Some of their work analyzing the teamwork aspects was published in [73, 84]. Research was also done by this team in the field of team reformation in case of agent's failure in order to improve the robustness of the system, such as helicopter being shot down by the enemy or situation changing in an unexpected manner. The results of this research were published in [74].

Another interesting domain where collaboration in a multi-agent system is desirable is the RoboCup soccer competition. Teams of simulated robots compete against each other in soccer matches. Good description of this problem domain can be found in [103]. The agents have to collaborate together in order to score a goal and prevent the opposing team from scoring. This domain is interesting from the research point of view, because it is clearly defined with fixed rules and easy to simulate. The problems faced by the competition participants are well described by Marsella et al. in [65], where the authors analyze the behavior and performance of two fielded teams.

In the United States, NASA is performing lot of research on collaborative systems with the focus on space flight. The research is done on training systems where humans are trained in simulated environments for complex assignments related to the spacecraft operation (such as the training system described in [72]). Another area where NASA is evaluating collaborative is live spacecraft operation. A life support control system based on collaborative agents and agent-human collaboration is described in [67].

A different type of collaborative agent was described by Doyle and Hayes-Roth in [19]. Their system is text-based, implemented using a MUD technology which originated in multi-user text adventure games popular in late 80-ties and beginning of 90-ties of the last century. The autonomous agent Merlyn is a wizard acting as a guide to the user visiting this environment. Since the environment is text-based, the environment designers are relieved from the tricky animation problems required to achieve a believable presentation of the agent.

2.4.2 Theories

Collaborative agents have been and still are subject of extensive research. There are several established theories and formalisms used to describe the collaboration process in the multi-agent system. The following list is not attempting to be exhaustive, however it tries to list the most important and influential works in this field:

- Joint intentions/commitments
- Joint actions/responsibilities
- Communicative multi-agent team decision problem
- Shared plans
- Delegation

Most of the works on collaborative agents have roots in the BDI[3] logic or assume that the agents are BDI controlled. BDI was introduced in 1987 by Bratman in [12]. He postulates that the behavior of an autonomous agent can be controlled by three main tenets *beliefs*, *desires* and *intentions*. Beliefs of an agent express its knowledge about the world and its internal state. Desires are high-level goals, such as a desire to stay alive or to not be hungry. Finally, the intentions pose short term goals for the agent which has to elaborate the ways how to achieve them – for example an agent can intend to go to the dentist, leading to planning process to work out how the intention could be satisfied.

In order to formalize the issues of collaborating agents and the related coordination issues, the theory of joint intentions/joint commitments was elaborated mainly by Jennings in his numerous publications. Jennings defines the principle of *commitment* and *convention* in [47].

In the BDI scenarios, an agent has frequently an intention to achieve something. Jennings argues, that what is important is actually the *commitment* to perform the task i.e. the agent could intend many things, but only if it is committed to perform them is it interesting. The commitments act also as a filter for possible actions – an agent will not commit itself to a task which conflicts with other commitments without good cause.

The notion of "good cause" when to abandon a commitment was formalized by *conventions*. The convention formally defines when is it admissible for an agent to abandon a commitment. Ideally, an agent should never abandon its commitment, however the circumstances may change and the agent may be forced to do so – for example to react to an unexpected threat.

The extension of individual behavior to multi-agent collaborative systems is based on the notion of joint commitments. The difference between individual commitment and joint commitment is that joint commitments concern more than a single agent. The basic concept is that in order to collaborate, the agents have to commit themselves (pledge) to perform the collaborative task.

The theory of *joint responsibilities* and *cooperation knowledge levels* defining how participants of the groups cooperating on solving a problem should behave in order to successfully achieve the goal was published in [46, 45]. Integration of the joint intentions inside of a BDI architecture was described by Jennings in [48]. He discusses how to implement a Belief, Desire and Joint-Intention system for collaborating agents.

[3]Belief-Desire-Intention.

The formal model of collaborative problem solving based on joint commitments and conventions was elaborated by Jennings and Wooldridge in several landmark works [112, 113]. Finally, the impact of agent-based software engineering with the emphasis on the collaborating agents was discussed in [49].

The joint intentions/commitments theory was a basis for many agent frameworks, such as STEAM or later TEAMCORE by the team of Milind Tambe. STEAM is a multi-agent simulation architecture based on the joint intentions theory. It was implemented using SOAR rules and publishes in [100], together with the typical domains – helicopter pilots and RoboCup soccer competition. Tambe further elaborates on the collaboration topic in [102], where he analyzes roles of the team members, possible team work failures and the recovery techniques.

In [86], published by Pynadath et al., the problem of building a meaningful team is analyzed. The premise is that in a typical multi-agent system, the agent have different capabilities, are built using different technologies and techniques. The authors analyze, how to effectively describe the capabilities of individual agents and how to utilize this information to form a team suitable for solving a task at hand. The result of this work is the concept of *team-oriented programming* (TOP) which was implemented on top of the TEAMCORE agents. The authors demonstrated successful team creation and cooperation among multiple agents given the specification of the desired goal and organization hierarchies.

The work [52] by Kaminka and Tambe describes a monitoring system for multi-agent systems to detect collaboration problems, particularly disagreements between the agents. The authors present the scalable YOYO algorithm for this purpose. Another paper [83] by Pynadath et al. tackles the monitoring problem by the concept of *overhearing* – non-intrusive monitoring of the agents by listening in to the routine communication between the agents and using a plan recognition approach to identify the ongoing activity.

In [106] Tambe and Zhang elaborate on the problems specific to persistent teams implemented using the STEAM framework. Persistent teams pose specific challenges compared to short-lived teams, such as hierarchy, adaptation to environment over time using the gained experience or establishment of standard operating procedures. The authors introduce teamwork model for persistent teams using the Markov decision process.

The complexity and optimality issues of the multi-agent teamwork process were analyzed by Pynadath and Tambe in [84]. The authors analyze the performance of the multi-agent system using the COM-MTDP model (communicative multi-agent team decision problem). This model allows evaluation of both the optimality of the teams performance and the computational complexity of the agent's decision problem. They present examples of using their model to analyze systems based on the joint intentions theory.

Grosz et al. described in [38] their work on *SharedPlans*. The work discusses how a shared plan among multiple participating parties can be elaborated during the discourse between the participants, taking into account their current mental state and their goals. In order for the two (or more) parties to successfully collaborate, several basic conditions have to be met (based on the joint intentions theory):

1. The parties have to have a common goal (for example to find an itinerary for a flight satisfying a set of constraints).

2. Have agreed on a sequence of actions (recipe) how the goal is to be accomplished.

3. Are each capable of performing their assigned actions.

4. Intend to perform their assigned actions.

5. Are committed to the overall success of the collaboration (not only their own part).

The shared plan idea was implemented by Rich and Sidner in their work on the COLLAGEN system [87]. COLLAGEN is a prototype to facilitate collaboration of an autonomous agent with human user, such as airplane ticket booking using natural language and shared plans approach.

The theory which is closest to the work presented in this thesis (collaborative agents based on delegation) was published by Ioerger and Johnson [44]. In particular, they formally define the hierarchical model of responsibility (who is responsible for satisfying a particular goal) expressing the delegation hierarchy. Afterwards the authors introduce the formalism for delegation – the *canDelegate* predicate describing the conditions when the task can be delegated. The main conditions are twofold: the delegate has to be *capable* of performing the delegated task and the delegating agent has to have some *authority* over the subordinate. This work provides formalism for describing existing systems using delegation-based collaboration – such as Open Agent Architecture (described in [68, 14]) and also this thesis.

2.4.3 Technologies

The efforts on collaborating agents produced many technologies to facilitate the exchange of information among the cooperating agents and to coordinate the team efforts. This section presents a short summary of the most influential technologies currently in use.

Blackboards

Blackboard systems implement an idea of shared information store accessible to all collaborating parties. Such shared store is called *blackboard*. In a typical blackboard system, the participating agents watch the blackboard for data they can process. Once such data are posted to the blackboard, the agent processes them and writes back the results for other agents to use or can alternatively post a query for other agent to answer. Frequently, there is no other coordination or negotiation going on among the agents apart of the sharing of the data posted to the blackboard.

One of the first blackboard systems published in literature was LINDA by Gelernter and then FLiPSiDE by Schwartz [94]. The latter describes a programming logic system for distributed expertise in financial domain. The system is used to monitor the state of the portfolio, to monitor the market situation and to plan and execute the trading operations.

Another well known blackboard system is ACORN developed by Marsh et al. [66, 114]. ACORN introduces concept of *café* where the agents can share information by a process the authors named *mingling*. The mingling process is essentially a key-phrase matching algorithm used to determine whether the shared information is of relevance to the participating agent. Another option to exchange information is by posting to the blackboard using a given set of rules.

Negotiation, conflict management

Negotiation is a key collaboration concept. The collaboration participants negotiate creation of the team, establish common goals and elaborate plans for team actions. Negotiation process is also necessary to reconcile differing views of the world by individual participants. There are several technologies which try to address the negotiation issue. They are frequently part of larger frameworks, such as STEAM.

CONSA (COllaborative Negotiation System based on Argumentation) described by Jung and Tambe in [50] tries to address the problem of intra-team conflict resolution in the STEAM multi-agent framework. The conflict resolution in the CONSA framework consists of four distinct stages: *proposal generation* by a team member, *opening*, *argumentation* and *termination* stage. The potentially conflicting fact is proposed by one team member. During the opening stage, each team member is expected to identify, whether the proposal conflicts with his beliefs or not. In the argumentation stage, the resolution of the conflict is proposed and the agents evaluate it, potentially continuing the argument with better proposals. In the termination stage there are several possible outcomes. If there is an accepted proposal for conflict resolution, the process terminates and the solution is accepted. Otherwise the process either fails if no solution was found and accepted or succeeds, if the conflict was identified as irrelevant.

Lander in her dissertation [60] elaborates a negotiation-based technique for distributed search, where an agent proposes a solution and other team members work on extending it further towards finding a global solution.

The Electric Elves system developed by Pynadath, Tambe et al. and published in [85] is a multi-agent scheduling system based on the Teamcore framework (descendant of the STEAM system). The users are equipped with personal agent proxies which maintain their schedule and current location using handheld mobile devices (Palm Pilots and GPS). The agents allow automatic coordination of activities such as scheduling meetings to be performed with minimal human involvement.

Partial global planning (PGP) by Durfee et al. (published, for example, in [20, 21]) originated in the domain of distributed sensor networks where the individual sensors represented as BDI-like agents have to efficiently collaborate together to establish a high-level interpretation of the sensed information. This process has to frequently occur in the presence of incomplete or noisy data and without a central coordinating authority which poses a single point of failure and potential performance bottleneck. The solution proposed by Durfee et al. is a system, where the individual nodes (agents) process the information in parallel and exchange abstract information in order to combine their local views of the situation into the global one.

Partial global planning brings together several techniques, such as *contracting*, *planning*, *organizing* and *result sharing* into a hybrid system where the individual agents are building their own views of the global state (partial global plans) and then use negotiation techniques to exchange them, propose changes and resolve coordination issues arising while solving the main problem.

Decker and Lesser created generalized partial global planning (GPGP) [16] algorithm by extending the original PGP into a whole domain-independent family of coordination mechanisms. These mechanism are selected for use depending on the task being solved. Another addition was the introduction of deadlines into the scheduling, specifying hard constraints until when the scheduled tasks have to be performed. Also the communication between the agents is happening at different levels of abstraction, leading to smaller amounts of information being exchanged compared to GPG where whole partial plans were exchanged each time.

Finally, there is an important standard relevant to the presented work – the FIPA[4] Contract Net interaction protocol published in [29]. This protocol describes a general negotiation process, where the agents are trying to elaborate a common conclusion – such as creation of a common action plan or bidding process to award contract for completion of a particular sub-task.

[4]Foundation For Intelligent Physical Agents, `http://www.fipa.org/`.

Agent communication languages

Multi-agent systems frequently employ agents developed using different technologies and standards. The resulting heterogeneous systems require a standardized means of exchanging information and requesting services among the agents, typically in the form of a *communication language*.

From the theoretical point of view, the link between language and social activity was made in *speech act theory* [95] using the concept of *illocutionary act* – a particular kind of social interaction performed in uttering of some sentence, such as promising, informing etc. One of the most important devices to determine the illocutionary force of the message in a particular (human) language is the lexicon of speech act verbs. In the case of English, such lexicon contains verbs such as *promise* or *warn*.

In the case of an agent communication language the counterpart of the speech act verbs lexicon is the catalogue of performatives (types of communicative actions) specifying the ones deemed important for the multi-agent system, such as *inform, query* or *request*. One such catalogue is the FIPA Communicative Act Library (CAL), available as a standard [28].

One agent communication language commonly in use is the FIPA Agent Communication Language [27] built on top of the FIPA Communicative Act Library and the associated standards, such as FIPA Interaction Protocols [30] and FIPA Semantic Language content Language [32].

Another commonly used agent communication language is KQML[5], described in [15, 57, 26].

KQML allows to manipulate existing knowledge of an agent by the means of received messages – requests to perform a certain action, such as to query a state of an agent or to notify an agent about a change. The transmitted messages have their type identified by *performatives*, such as `ask-one`, `tell`, etc. denoting the primitive message type.

KQML is content agnostic, however it is commonly used together with the FIPA KIF[6] content language standardized in [31]. Both languages are popular choice for sharing and exchanging knowledge among the agents in the multi-agent systems. KQML and KIF inspired many developers of multi-agent systems (e.g. RETSINA [99]) and KQML and KIF-like syntax is commonly used with various extensions.

An effort to provide a standardized method to design new agent communication languages was presented by Serrano et al. in [97, 96]. They present a meta-model based on the analysis of the relationships between role of the agent, interaction between agents and the associated communicative action. This meta-model is known as RICA (role/interaction/communicative action). New agent communication languages can be produced from the RICA meta-model by specialization.

Delegation

The concept of task delegation as a collaboration technique in multi-agent systems is known for a long time. It mirrors closely the hierarchical management/command structures common in armed forces or organizations where the responsibility for the task is delegated from a superior to the subordinate (delegate).

Compared to the contracting approaches (such as Contract Net), task delegation does not have to employ negotiation process (such as bidding). The task is delegated directly to one or multiple agents which become responsible for its completion.

From the theoretical point of view, Ioerger and Johnson described in [44] a formal model of responsibilities in the multi-agent systems with the emphasis on systems allowing task delegation.

[5]Knowledge Query and Manipulation Language.

[6]Knowledge Interchange Format.

One of the main ideas with relation to delegation is the concept *when* is it possible to delegate a task – if and only if the delegating agent has some form of *authority* and the delegate *is able* to perform the delegated task in some way (even by sub-delegating it further).

One of influential works in this area is Open Agent Architecture (OAA) developed at SRI during the nineties of the last century. There are many publications describing this framework, for example [14] or more recently[68].

The central concept in the Open Agent Architecture is the *facilitator*, providing services to other agents. The original OAA facilitator serves as both matchmaking and data storing service, brokering communication in a distributed collection of autonomous agents. It is also able to subdivide larger tasks to smaller pieces to be resolved by multiple agents.

TAEMS[7] framework was developed by Decker [17] as a mean to quantitatively describe and reason about the interrelationships between tasks. In case that the tasks are worked on by differing agents, the relationships are *coordination relationships* and are important to the design of the coordination algorithms.

This framework represents the structure of the task at multiple abstraction levels. The highest level is a *task group* which contains all tasks having explicit interrelationships. A *task* is a set of lower level sub-tasks or executable methods. The components of task have explicitly defined effects on the outcome of the encompassing task. The lowest abstraction level are executable methods – schedulable entities, such as piece of code with its data, fully instantiated plan or even an instance of human activity.

TAEMS tries to be independent from any particular agent structure, the only basic properties required from an agent are beliefs (state) and actions (executing methods, sensing, communicating).

The TAEMS framework provides facilities useful for agent coordination by defining how agent actions and tasks relate to each other, as well as a calculus for describing various plan alternatives that can be chosen at run-time by an action scheduling component.

RETSINA[8] is a multi-agent framework created by Sycara et al. [99]. The architecture has three types of agents – information agents providing access to data stores, interface agents tasked with communication with the user and task agents aiding user (or other agents) perform tasks by formulating problem-solving plans and executing them using communication and information exchange with other agents.

Agents in RETSINA are distributed and directly activated based on the top-down elaboration of the current situation – as opposed to e.g. Open Agent Architecture, where this activation is performed indirectly by the brokering agent/facilitator. Agent locating is performed by matchmaking agents acting as yellow pages – directories of agents advertising certain capabilities. The organizational structure is formed dynamically, dependent on the needs of the task being solved.

Planning in RETSINA is achieved using hierarchical task networks (HTN). Each agent has a current set of goals, current set of task structures and a library of task reduction schemas. The task reduction schema specifies how to perform a task by prescribing a sequence of sub-tasks/actions to be performed and the corresponding information-flow relationships between them (e.g. passing the output of one task as an input to the following task).

DECAF[9] is an agent toolkit created by Graham et al. [36]. The framework provides the basis for design and implementation of multi-agent systems usable by non-researchers by providing specialized tools (such as the Plan Editor) to program agents.

[7]Task Analysis, Environment Modeling, and Simulation.

[8]Reusable Task Structure-based Intelligent Network Agents.

[9]Distributed, Environment-Centered Agent Framework.

Planning in the DECAF framework is managed using the hierarchical task networks (HTNs) similar to the RETSINA framework. The agents consist of set of potential objectives or goals and a collection of actions that may be planned and executed to achieve these goals. The goals are represented as complete task reduction trees (HTNs) using the annotations from TAEMS task structure description language. The leaves of the tree are basic agent actions (primitive tasks). The complete plans to solve the goal may contain hundreds of the primitive actions and are created using the Plan Editor.

Communication between the agents in the DECAF framework is achieved using the KQML agent communication language. Legacy systems unable to communicate in KQML are wrapped using proxy agents to connect them to the framework.

Another delegation-based multi-agent system used for military simulations is MOKSAF, described by Payne et al. in [78]. The goal of the MOKSAF architecture is to explore collaboration in mixed human/agent teams. The framework makes use of the Matchmaker, which is a special agent matching the incoming requests with the services advertised by the other agents and KQML to communicate among the agents.

2.5 Action planning

The issue of action planning is closely related to the problems of autonomous problem solving. The goal of planning is to find a sequence of actions transforming the given initial state of the system into the desired goal state.

Planning is a difficult and often intractable problem, if the problem being solved is large. Therefore there were attempts to simplify the problem. One of the alternative approaches is GOLOG, created by Lespérance et al. and described in [61]. GOLOG attempts to be a middle ground between low-level imperative programming and high-level planning-based solution. It is a specialized programming language permitting creation of agents where many problems conventionally left to planners are addressed using non-determinism.

A similar approach to GOLOG was used by Funge in his work [33], where he introduced cognitive modeling language (CML). CML allows high-level description of the goal for the virtual character, but letting it automatically search for suitable action sequences in order to satisfy it.

Another common approach how to reduce the complexity of the planning is to use a hierarchical planning system. The high level plan is typically generated by an agent in the upper parts of the hierarchy and refined/specialized by the agents in the lower ranks responsible for their particular tasks. The advantage is that the high level agents do not have to plan down to the details required by the low level agents for performing the assigned task. One example of such hierarchical planning system was described by Baxter and Hepplewhite in [9], where it was used to drive a military simulation.

Another application of hierarchical planning in the teamwork context in the multi-agent simulation system was demonstrated by Alonso and Kudenko in [6]. They describe an application of STRIPS-like planning in a military simulation.

The next sections will attempt to provide an overview of the main approaches to action planning and action selection used in development of autonomous agents. The techniques can be divided into two main groups, each of them will be described separately.

2.5.1 Iterative techniques

The main trait of iterative techniques is that the decision making process to identify next action(s) to take is repeated during each simulation frame of the agent. Alternatively, the plan can be partially created at the beginning and iteratively refined and updated according to the current situation.

BDI

BDI agent model was mentioned in the section 2.4.2. From the planning point of view, BDI agents are typically iterative in nature – they make their decisions during every simulation frame and use the library of pre-created plans to address various situation which can arise during the lifetime of the agent. BDI agents using propositional planning instead of the pre-built plan libraries are possible (such as the work of Meneguzzi et al. [71]), but are not common.

Examples of BDI agents using iterative action planning are well known, for example KGBot for Unreal Tournament by Kim [53] and coalition forming collaborative agents described by Griffiths and Luck in [37]. Another famous applications of iterative BDI agents are games like The Sims and Black&White by Electronic Arts.

Rule-based systems

Rule-based (also known as production) systems are a popular technology for implementation of autonomous agents. They consist of three main parts:

- Working memory
- Rules
- Decision mechanism

The working memory contains a short term knowledge, representing the internal state of the agent and its beliefs about itself and the surrounding environment. The rules define the basis of the behavior of the agent, usually in the form **`if`** `precondition` **`then`** `effect`. The decision mechanism is used to decide which of the applicable rules (rules, where the precondition is satisfied) is to be fired.

One well known example of a rule-based system is SOAR[10], originally developed by Newell in the late 1980s and published in [59]. SOAR goes beyond a simple rule-based system which just selects and fires rules. It is able to reason about the operators (actions to take) themselves in order to identify the best one to apply and it is able to automatically generate sub-goals for resolving impasses, whenever it is unable to make decision directly.

SOAR is also frequently interpreted as a model of human cognition and used for cognitive modeling. Another view of SOAR is as a specialized programming language for artificial intelligence.

SOAR is used as a basis of many multi-agent simulation frameworks, such as TEAMCORE[105], STEAM[101] and many others. In [102], Tambe describes how a SOAR architecture can be used for creation of agent teams using team operators.

Other than that, SOAR was used also in game environments, such as the famous SOARBot which was used together with the Quake II engine [58], Unreal Tournament and Descent III games to simulate a computer opponent for a human player. SOAR was also frequently utilized in the RoboCup competition to implement agents simulating a robotic soccer team.

[10]`http://sitemaker.umich.edu/soar`

ItPlanS

The ItPlanS planner is a special case between iterative systems on one side and propositional planners on the other. It was developed by the group of Norman Badler at the University of Pennsylvania and deployed in the famous SodaJack system. The planner was published many times, for example in [34, 63].

SodaJack is an application of a Jack animation system developed at the university of Pennsylvania to a simple domain, where a virtual human character is manning a soda stand and is expected to dispense products to visitors. The Jack animation platform is driven by a hierarchical planing system, utilizing three different planners – ItPlanS as a high level planner, one specialized planner to locate relevant objects and knowledge and a specialized planner for object specific reasoning.

ItPlanS planner works in an iterative manner – building and expanding the produced plan as required, depending on the results of the previous actions and the current state of the agent's environment. For example, if the agent is asked to serve soda, it has to retrieve the cups first. The cups can be in different places – such as in the cupboard, on the counter, etc. The planner first generates a task to search for the cup in the cupboard (for example). Once the cup was found, a new plan for next action is generated to continue towards the goal. If the cup was not found, a new search task is generated to search elsewhere.

The advantage of the ItPlanS approach is that the system can be very responsive, there is little delay caused by the planning. At any given moment there is a plan of what to do, even though it may not be complete (fully specialized). The disadvantage of this approach is that the plan is never completely known in advance, potentially leading to poor behavior. For example, the agent will first search for the cup, finding it in the cupboard. Then it realizes that an ice-cream scoop is also required and will start searching for it again, inspecting the cupboard from scratch instead of retrieving both the cup and the scoop together the first time.

Procedural Reasoning System

Procedural reasoning system (PRS) developed by Georgeff [43] is a BDI-like reasoner merging the advantages of reactive and goal-oriented reasoners. PRS also features guaranteed response times which are important in real-time control systems.

The information in PRS is represented by a set of beliefs, set of current goals, library of plans/procedures containing sequences of actions called *knowledge areas* (KA) and an intention structure containing a partially ordered set of plans selected for execution.

The interpreter manipulates these structures – it selects the appropriate KA based on the beliefs and goals, puts them into the intention structure and finally executes them. Selection of the KAs may be triggered either because of some event in the environment activating the *triggering* part of the KA (a precondition saying when to trigger the KA) or because the KA provides solution to some of the current goals. This property ensures fast response to changes in the environment while allowing means-end reasoning for problem solving.

The procedural reasoning system was deployed in several real-world applications, such as fault isolation in the space shuttle propulsion systems. Another application of PRS was network traffic management for telecommunication networks, where the reasoner helped with managing congestions and network disruptions by re-routing traffic.

The advantage of PRS and PRS-like systems is the combination of quick response (reactivity) to external events, similar to rule-based systems while retaining the capability for means-end reasoning.

The disadvantage of the techniques using fixed plan libraries is their extreme domain-specificity. The plan libraries encode a domain-specific knowledge, such as the correct fault isolation procedures. However, if the reasoner does not have a suitable plan available in the library, it will be unable to solve the problem, even though it is solvable by a correct (but unknown to the reasoner) sequence of primitive actions.

2.5.2 Propositional STRIPS-like planning

Propositional planning is a very old topic in the artificial intelligence field. It is being researched since early seventies of the twentieth century, with many works published. Propositional planning found many applications in robotics, autonomous agents, industrial process control, spacecraft control and many others.

In propositional planning the problem is described in terms of predicates representing assertions about the world state. The possible actions are defined in terms of *operators*, using notions of *preconditions* and *effects* to describe when it is possible to perform them and what are their effects on the world state.

The planning problem (transforming the initial state to the desired goal state by a sequence of operator applications) is defined using the initial state of the world, the desired goal state and the permitted operators. This approach was pioneered by Fikes and Nilson in their seminal paper [25], which described the STRIPS planning system driving the SHAKEY robot.

During the past decades, many STRIPS-like planners were developed. The original system was extended over time to allow things such as quantifiers, conditional effects, basic arithmetic, uncertainty, temporal planning and many others. Many algorithms were developed, among the most notable ones – UCPOP [79], Prodigy [109], Graphplan [10]. Most of these planners spawned whole families of derived planners, for example Sensory Graphplan [7, 111], FastForward and Metric FastForward [41, 42], AltAlt [76] and many others are all derivatives of the original Graphplan planner published by Blum et al., using the same planning graph data structure albeit in different ways.

Of particular relevance to this thesis are two planners from the Graphplan family – Sensory Graphplan by Weld et al. and Metric FastForward by Hoffmann. These two planners were used as a part of the multi-agent simulation framework described in the chapter 6.

Propositional planning is highly complex task, typically exponentially increasing in complexity with the amount of information the planner has to process during the planning process. To lower the load on the planner, it is possible to remove the facts which are not relevant to the problem being solved by pre-processing. Such approaches were described in [22] by EL-Manzalawy and by Nebel et al. in [75].

In order to be able to compare the performance of different planners the need for standardized benchmarks was identified. These benchmarks are standardized problems to be solved by the planners, both synthetic and realistic. To be able to use the same set of problems with all planners, the planning community created common planning domain description language (PDDL) which is supported to various degrees by most of the propositional planners available. PDDL evolved through several versions, from the original created by McDermott to the most recent version 2.2 and the a new language Opt. The specifications for all these languages are publicly available at [69].

The main differences between different PDDL revisions are capabilities to model different types of problems which were added or removed – such as durative actions (actions having continuous effect over a period of time) or handling of numeric fluents.

The feature subsets of PDDL important for this thesis are STRIPS (containing STRIPS-like operators) and ADL, which includes also disjunction and quantifiers in preconditions and goals and also conditional and quantified effects.

Graphplan

Graphplan is a very influential planner developed by Blum and Furst and published for the first time in [10] in 1997. Graphplan had a very good performance compared to the other planners at the time when it was developed (such as UCPOP and Prodigy) and many planners were derived from it.

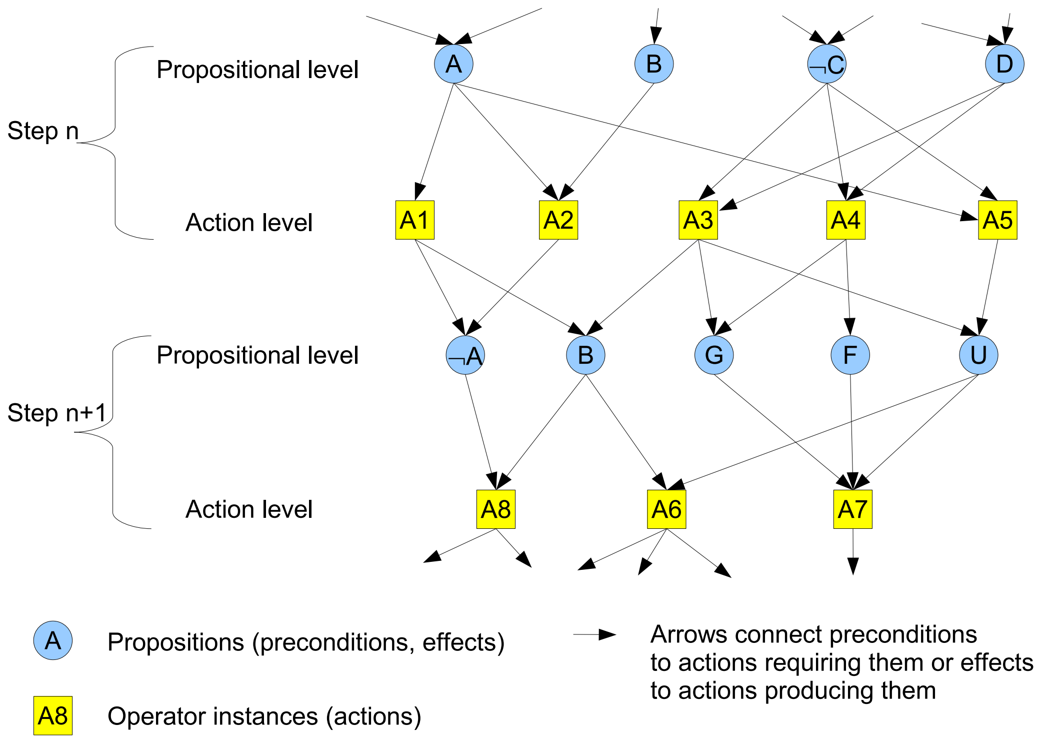

Propositional level
Step n
Action level
Propositional level
Step n+1
Action level
A
B
¬C
D
A1
A2
A3
A4
A5
¬A
B
G
F
U
A8
A6
A7
A
Propositions (preconditions, effects)
A8
Operator instances (actions)
Arrows connect preconditions to actions requiring them or effects to actions producing them

Figure 2.3: Planning graph structure

Graphplan works by analyzing a data structure named *planning graph*. A planning graph is a leveled graph where levels consisting of proposition nodes and action nodes alternate (see fig. 2.3). The first level is propositional one, with propositions describing the initial state of the world. Next level is action level, where all possible actions (with preconditions satisfied by propositions of the previous propositional level) are present and connected by edges with the propositions satisfying their preconditions. Next propositional level is created from the effects of the actions. The planning graph in figure 2.3 is simplified, the required no-op actions propagating effects unchanged from level to level are omitted for readability.

The graph building process is repeated until the graph *levels off* – two consecutive propositional levels are identical – or the last propositional level contains all propositions needed to satisfy the goal and they are not mutually exclusive (mutex).

The search for the plan is performed as backward-chaining search starting at the last propositional level and searching for actions producing the required propositions. The search then progresses to the preconditions of these actions in a previous propositional level and again actions are searched for which satisfy these preconditions. The search finishes when the first propositional level (initial state) is reached and the path to get there is the plan.

Original Graphplan produces partial-order plans, where the plan is divided into separate time steps with fully specified order, but the actions inside the time steps are unordered. The order of actions in the single step is not defined and they can be executed in arbitrary order because they are non-conflicting (non-mutex, it is ensured by the planning algorithm).

The speedup of Graphplan compared to the UCPOP or Prodigy comes mainly from propagation of the mutual exclusion relationships between the propositions and actions. Obviously, if two propositions are mutex (e.g. $A \wedge \neg A$), then two actions which require these two propositions to hold at the same time for their preconditions to be satisfied cannot be executed in the same step – they are mutex as well, because they have conflicting needs. Another criterion used by Graphplan to determine whether two actions are mutex is to test whether an effect of one removes an effect of the other – in that case the effects are conflicting and the actions are mutex.

An important technique used by Graphplan to speed-up the search for the plan (the most time intensive part, the graph building is fast) is *memoization* – a dynamic programming technique used to remember the combinations of nodes at a particular level which pose a dead end for the search. It helps the search to avoid the already explored branches not leading to the goal.

Sensory Graphplan

Sensory Graphplan (or SGP) is a partial order planner produced at the University of Washington as an extension to the classical Graphplan. Its design was published in [7, 111]. SGP is a planning graph based planner (property inherited from Graphplan) which extends the classical STRIPS-like Graphplan to handle uncertainty and sensing actions.

Classical STRIPS-like planners handle the predicates expressing the current state of the world in a strictly binary manner – the predicate evaluates to either true or false. Sensory Graphplan allows the predicate to be defined as *uncertain* (unknown), enabling an escape path from the problems caused by the closed world assumption – if a predicate is not defined inside the agent's state it does not have to mean that it is false but only that no information about it is not available.[11]

SGP uses the information about uncertainty of predicates to perform two actions during the planning process:

- Insert a sensing action intended to determine the truth value of the uncertain predicate at the execution time.
- Fork the planning graph into two parallel worlds, depending on the outcome of the sensing action (one branch for the "true" case and the other for the "false" case).

Plans generated by SGP are known as *conditional* or *contingent* plans. According to [91],

"Conditional planning deals with bounded indeterminacy by constructing a conditional plan with different branches for the different contingencies that could arise. Just as in classical planning, the

[11]Strictly speaking, this would mean that SGP is using open world assumption. This is not completely true because of the implementation of the planner derived from standard Graphplan. If a predicate in the initial state is not defined and neither declared explicitly as *uncertain*, the planner will not find the corresponding node in the planning graph during the search and it will treat it as being false.

agent plans first and then executes the plan that was produced. The agent finds out which part of the plan to execute by including *sensing actions* in the plan to test for the appropriate conditions."

The generated partial order plan carries additional context information with each planned action to enable the agent to verify the conditions determining which branch of the plan to execute. In case of SGP planner, the actions carry information about the planning worlds (states) where the action *must not* be executed.

Uncertainty and sensing are very common issues in VR domains and their support directly in the planner is a very useful property – e.g. an agent is able to "discover" that the state of a door needs to be checked before attempting to open it (and potentially failing to do so if the door was open already). Such problem can be handled easily by the planner if the state of the door is declared as *uncertain*. The planner will also generate the corresponding sensing action in the plan.

Another advantage of Sensory Graphplan is the fact that it generates partial order plans. Partial order plans allow for sets of mutually independent actions to be executed without defined order (even in parallel). Such partial ordering is important when working with teams – in practice, it is desirable that the team leader dispatches the orders and the independent actions are performed in parallel, not sequentially. For example, moving twenty team members sequentially to a distant location will take a lot of time, whereas a partial order plan may call for moving them there in a single planning step – the actions are independent (non-mutex, the planning algorithm ensures this by the way of how the planning graph is traversed when searching for the plan) and can be scheduled in parallel.

The large disadvantage of Sensory Graphplan is its general slowness and resource consumption. SGP uses the same brute-force backwards chaining search algorithm as standard Graphplan but adding the extra complexity of conditional planning graphs, essentially doubling the size of the space to be searched for each uncertain predicate (exponential complexity). This limits its usability to simpler problems, such as the "Virtual guide" scenario described in section 7.2.

Metric Fast Forward

Metric-FF (or Metric Fast Forward) is a total order planner developed by Jörg Hoffmann from Max Planck Institute in Saarbrücken, Germany. The planner is based on planning graphs as well (same as SGP and Graphplan), however there are few important differences. Metric-FF was a top performer in the numeric track of the 3rd International Planning Competition organized in 2002. The planner is based on an older FF planner, described in [42]. Metric-FF itself was published in [41].

Metric-FF extends the original STRIPS-like FF planner to handle ADL domains and arithmetic – supports features like typing, disjunctive preconditions, equality, quantifiers, conditional effects and expression evaluation. The current implementations of both Metric-FF and the agent framework do not support/use all of these features, but equality, conditional effects and expression evaluation are used to achieve a more realistic simulation.

The FF family of planners works by building the planning graph in a similar manner as the original Graphplan, however the search phase is completely different. Graphplan uses a backward-chaining search producing a partial-order plan, whereas the FF family uses an A^* algorithm to search from the initial state towards the goal state (forward search).

The A^* algorithm uses a heuristics to guide the search towards the most promising nodes of the graph first, strongly increasing the chance that the goal node will be found without having to do a full graph search. In the FF and Metric-FF planners the heuristics is computed using a simplified version of Graphplan, which does not consider the del-effects of the actions. To compute the value of the heuristics, Graphplan is invoked from the current state and builds the planning graph toward the goal state. The length of the resulting planning graph (number of planning steps until the goal state appears in the planning graph) is used for the value of the heuristics, giving a lower bound on the needed plan length to achieve the goal state.

The huge advantage of using the Metric-FF planner is the raw speed and low memory consumption. On large problems, Metric-FF is typically orders of magnitude faster than SGP. Another advantage is the support for numerical variables and basic arithmetic, allowing to easily plan for a car with a limited amount of fuel or to simulate BDI-like states of an agent (e.g. thirst) and to plan with them (e.g. to decrease thirst the agent needs to drink).

Metric-FF is a total order planner, meaning that the ordering of the actions in the resulting plan is completely specified and fixed. This poses practical problems for handling teams of agents, because if the order is to be respected, all actions have to be executed sequentially. Taking the example from the previous section, an member of a team of 20 agents could start moving only after the previous one arrived to his destination, which is obviously not desirable (1 minute real time per agent to get in position means 20 minutes total for just the movement of the agents). The solution is to allow limited parallelization of the plan in certain special cases, where it is known that no inconsistencies will be created.

Such limited parallelization process was used in the “City riot” scenario described in the section 7.6. Because the `move` and recruiting actions were known to be non-interfering, it was possible to reorder the plan in such way, that the team members are recruited at the beginning of the plan, then the movement of the agents (policemen in this case) is performed in parallel and then the rest of the plan is done sequentially again. However, this is not a generic solution and does not work in every case. Conversion of a total order plan into a partial order plan is not always possible in general case and it is a hard problem in itself.

Chapter 3

Symbolic representation of a virtual world

In order for the machine to be able to meaningfully reason about the virtual world, it has to be encoded in a suitable form. What is typically available in a virtual reality simulation system is usually too low level for meaningful reasoning – for example geometric data (e.g. meshes) or animation sequences, either as keyframe animations or procedurally generated.

To enable the desired reasoning capabilities, the system designer has to provide a higher level symbolic information about the parts of the system. This information can be expressed in many different ways, however for simplicity and to be able to harness already existing tools, this thesis will focus on data representation derived from the propositional calculus and first order logic. This choice follows also from the intended focus on automated problem solving using propositional planning where the planning domain description has to be provided in this form.

First order logic is a mature field and its applications in artificial intelligence are well known, therefore the described applications will assume certain level of reader's familiarity with the topic.

3.1 General problems

Virtual worlds can range from very simple containing only few geometric objects to extremely complex, e.g. simulating artificial life. Regardless of the complexity, there are few general problems which are common to all of them if they have to be represented in symbolic form.

3.1.1 Environment representation

First of all, the state of the virtual environment has to be represented in a symbolic form. One of the common forms is to use predicate calculus. To meet the requirements of this work, several extensions will be also described here.

The definition of predicate calculus is quite extensive and well known. Informally, the predicate calculus is a representation language with a formally defined syntax, formal semantics and a set of sound and complete inference rules. The full definition was omitted for brevity, but can be found for example in [64].

In the virtual environment (world), we have several kinds of information that needs to be somehow represented:

- State/property of an entity (e.g. “the sky is blue”)
- Qualitative relationships between entities (“place A is connected with place B”)
- Quantitative values (“agent Martin has two teammates”)
- General rules holding in the world (“agent cannot be in two places at the same time”)

To represent this information, several approaches will be used. For the first two points, the natural choice is to use *predicates*. Predicates are usually written in literature in the form $blue(sky)$, where $blue$ is a predicate symbol and sky is an entity for which the predicate holds.

For the purposes of this work, another notation will be used: `(blue sky)`. The format is one of a tuple of symbols, where the first element of the tuple denotes the predicate symbol and the remaining elements are arguments. The reasons are mainly technical, tuple representation is easier to process by common tools.

The quantitative values cannot be expressed by predicates because these can have only two values – either true or false. To express a property which can change in different situation and is represented by a numeric value (e.g. amount of energy of a robot or the number of teammates), we need to use the concept of *numeric fluents*. Numeric fluents are expressed in the form of n-ary functions over the real numbers domain ($Object^n \rightarrow R$) associating the objects with the numeric value. An example could be `(THIRST)`[1] function returning the measure of simulated thirst for the current agent or `(distance ?x ?y)` returning the distance between two points.

The current state of the world can be represented as a set of predicates assumed to be true and a set of numeric fluents with their current values. For simplicity, a technique known as *closed world assumption* will be used – essentially any predicate missing from the predicate set will be assumed to be false. This technique has an obvious drawback – there is no way how to distinguish between an information which is *unknown* and *false*.

However, for the representation of the global state of the world this is sufficient, because the state of every object and agent has to be defined. In case of local beliefs of individual agents this convention has a consequence that if the agent has incomplete information (partial observability), it will assume the missing predicates to be *false*. This fact has an impact on agent design such as the necessity to ensure that sensing retrieves sufficient information from the environment before attempting other activities (e.g. planning) because otherwise the actions could fail due to missing information.

Finally, what is missing is the expression of the “rules of the game” or axioms which have to always hold in the virtual world. Typically, axioms express a fundamental relationship between predicates – for example an agent can be near an object, which implies that it has to be in the same room as the object. Thus, when the agent moves to another room, the predicate for being near has to be falsified as well. In practice, such rule can be written as:

$$\begin{aligned}\forall agent \forall object \forall place(near(agent, object) \wedge at(object, place) \wedge \\ \neg at(agent, place) \Rightarrow \neg near(agent, object))\end{aligned}$$

The implication holds in only in one direction. The reason is that the predicates $at(agent, place)$ and $near(agent, object) \wedge at(object, place)$ could be resulting from the effects of *multiple* different activities of the agent. In general, from the fact that the agent is not near to some object it is not

[1]All capitals notation, because the function directly returns a state variable. To make it easier to distinguish predicates from state variables, the latter will be written in all capitals.

possible to conclude that the agent is not in the same room as the object – it may be, but it didn't approach the object (and therefore the $near(agent, object)$ predicate evaluates to $false$). The purpose of the axiom is static verification of the consistency of the given world state and not to ensure that the agent's activity was performed properly.

The same axiom can be written in the PDDL notation:

```
(implies (and (near ?agent ?object)
              (at ?object ?place)
              (not (at ?agent ?place)))
         (not (near ?agent ?object)))
```

The PDDL version lacks the universal quantification. This is not an error, it is intentional. The universal quantification is implicit here because of how the expression is evaluated – the evaluation process tests all possible substitutions for the variables `?agent`, `?object` and `?place` (instantiations of the axiom) by default, without having to explicitly specify it. The process will be described in more detail in the section 6.

3.1.2 Partial observability

The virtual world is represented using the techniques described in the previous section. Using the predicate calculus it is possible to model the state of the environment. However, the symbolic representation as described above does not take into account the possibility that the individual agents living in this environment may have differing views of their surroundings. The shared global state would mean that all agents have the knowledge about everything in the world – the world/problem would be *fully observable*. Such representation does not faithfully model the reality, there is no reason why one agent has to know about the properties of an object never seen before or not even known to exist.

To address this issue, a two level data hierarchy is used, as seen in figure 3.1. There is a global state of the world, which consists of the set of predicates and numeric fluents representing the current situation of the virtual environment. This information is assumed to be always valid, it forms a reference basis for all further operations.

The second level consists of the *beliefs* of the individual agents. The beliefs are again a set of predicates and numeric fluents representing the state of the environment but from the viewpoint of the agent. This distinction is important, because beliefs may be incomplete (if the agent never encountered some object) or even outright incorrect (stale information – for example because another agent or user changed the world state without the agent in question being aware of it).

Such problems with the possible incompleteness or staleness of the information have to be addressed by the agent implementation – for example by providing sensing capabilities, cognitive abilities or by implementing communication between the system components such that the agents can notify each other about the changes.

Finally, the figure 3.1 shows also the user on the same level as the remaining agents, perceiving the world only as partially observable. For the experiments in human–agent collaboration this is interesting because the user should have comparable capabilities to the agents he/she is collaborating with. If the user has "god-like" skills, the whole concept of collaboration is meaningless – why to collaborate with an agent if the user can achieve the task using his/her special powers not available to others.

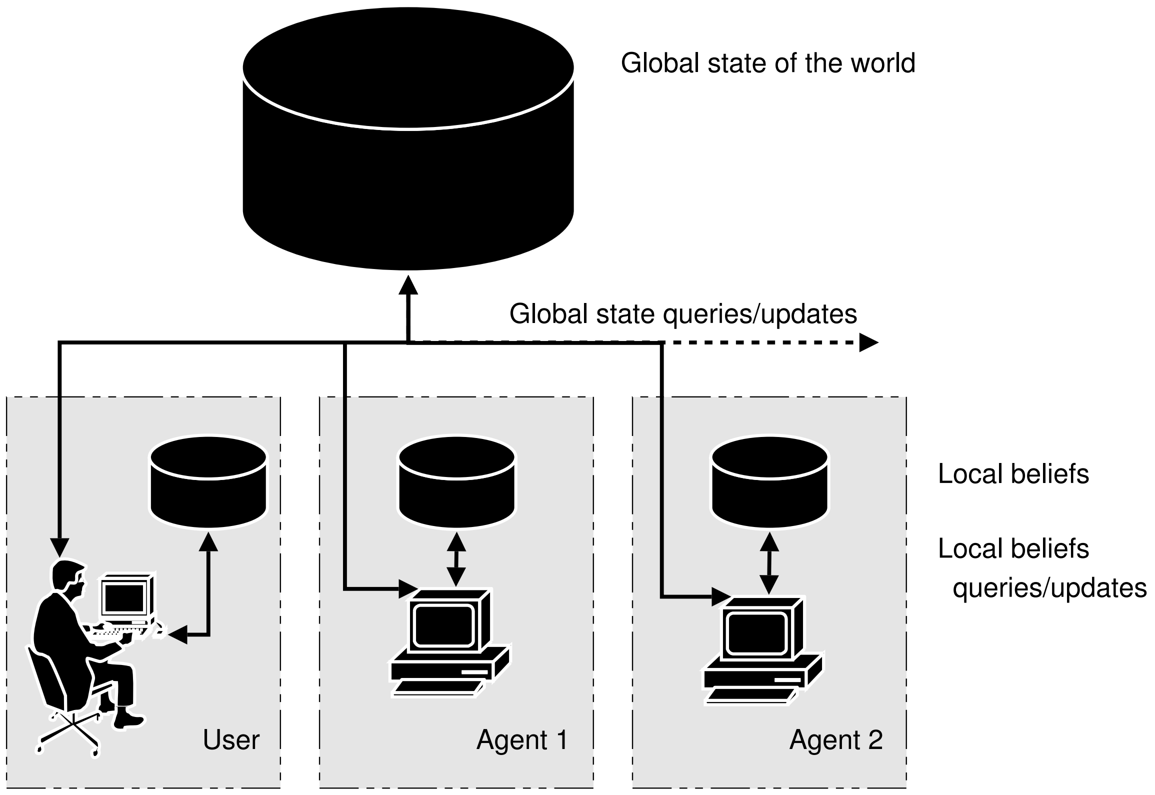

Global state of the world
Global state queries/updates
Local beliefs
Local beliefs queries/updates
User
Agent 1
Agent 2

Figure 3.1: Two level information organization

3.1.3 Task representation

To be able to meaningfully collaborate we need to symbolically represent tasks given to the collaborating agents. A naive way of achieving this is to use an imperative approach – "move to the room X" or "bring object Y". The imperative variant of task representation usually takes a form of *delegated action* which will be described in more detail in the section 4.2.

The imperative task specification typically looks as in figure 3.2. Each task specification is a tuple consisting of at least two components – a keyword (also known as *functor*) denoting the action to be performed and the name of the agent which has to perform it. If there are more than two components, the remaining components are parameters for the task. The actions are primitive and cannot be decomposed further (as opposed to e.g. HTNs in RETSINA).

```
(move A X Y)         ; asks the agent A to move
                     ; from place X to place Y

(drink gino coffee)  ; tells the agent gino
                     ; to drink the coffee he holds
```

Figure 3.2: Imperative task specification example

This approach is valid, however there is a major limitation – by using an imperative form, the agent is told not only *what* to do but also *how* to do it – keywords "move", "bring", etc. are tied to the specific animation(s) such as walking. That is not always desirable because it limits the possible ways of solving the task by the agent.

Often a more practical approach is to *declaratively* specify a desired final state. This can be done by logical expression containing predicates describing the desired goal state. In this way the agent is being told only what to do but not how. Essentially, it is equivalent to saying "I do not care how you do it but do it". In the most common case, some planning system is used to create a sequence of actions (plan) leading to the desired goal state.

If we wanted to express the same tasks as the two shown in figure 3.2 in a declarative way, we need to specify the desired state instead. Of course, only the relevant parts of the state need to be specified, not a full set of predicates and numeric fluents. An example is given in figure 3.3

```
(at A Y)            ; we are asking the agent A
                    ; to be at place Y

; We are asking agent gino to get rid of the coffee
; but in such way that he shouldn't be thirsty anymore -
; drinking eliminates thirst. This ensures that the agent
; will not simply put the coffee away but drink it instead.

(and (not (have-consumable gino coffee))
     (not (thirsty gino)))
```

Figure 3.3: Declarative task specification example

The second example is very simple, however at the execution stage a very complex behavior can be generated – e.g. in order for Gino to get his coffee, he has first to summon a waiter, make an order, the waiter has to go prepare the order and bring it back. Only after all this happens, is the agent able to consume his coffee.

If we wanted to produce the same behavior in an imperative fashion a whole plan would be necessary, specifying exactly each step to be taken. It would also have the disadvantage that actually the user/agent defining the task may not even be aware of what are the proper or possible actions to perform. The advantage of using the declarative task specification is very clear here.

The disadvantage of using declarative task specification is that the task may be performed in a non-obvious or unpredictable manner. The execution of the task may lead to un-anticipated side effects (such as the waiter from the example above moving objects around or depleting some scarce resource). Also, the goal specification may be non-intuitive if the desired behavior is complex. These problems have to be mitigated by careful design of the actions.

3.1.4 Action semantics

The final part of the data representation puzzle is the representation of the possible actions themselves. Actions were mentioned already in the previous sections, however not their representation.

The usual way how to approach the problem of representing the possible actions of the agents is to use the approach pioneered by Fikes in his STRIPS planner (see [25]). His ideas were generalized later and incorporated into the PDDL specification.

To represent the possible actions, every interaction of the agent with the environment has to be described in terms of its preconditions and its effects on the state of the agent/object when the action is performed. Fortunately, such information can be easily encoded using the situation calculus. A detailed description of it and its use to describe possible actions by an intelligent agent can be found e.g. in [91].

For the purposes of describing the semantics of the actions we need three axioms of situation calculus. The *possibility axiom*:

$$Preconditions \Rightarrow Poss(a, s)$$

determines *when* is it possible to perform an action. The predicate $Poss(a, s)$ denotes that the action a is possible to execute in the situation (state) s. The *effect axiom*:

$$Poss(a, s) \Rightarrow Changes\ that\ result\ from\ taking\ action$$

denotes *what* happens when the action is executed. The change to the situation s after executing action a is denoted as $Result(a, s)$. Finally, in order to resolve the representational frame problem, the *successor-state axiom* is used:

$$\begin{aligned}&Action\ is\ possible \Rightarrow\\ &\quad (Fluent\ is\ true\ in\ result\ state \Leftrightarrow Action's\ effects\ made\ it\ true\\ &\qquad \vee\ It\ was\ true\ before\ and\ the\ action\ left\ it\ alone)\end{aligned}$$

The successor-state axiom defines that the fluent (predicate) becomes true in the result state if and only if the right side holds as well. The successor state is therefore fully defined as a function of the current state and the action being performed.

This is a different situation than the axioms used to ensure the state consistency in section 3.1.1. There the implication holds only in one direction, because the state could have been modified in arbitrary amount of steps (actions), not just a single one as with successor-state axiom. Alternatively, the change could have been a consequence of the activity of another agent and was only perceived by the agent applying the axiom to its beliefs.

In our case each possible action can be formally expressed using the situation calculus, as shown in figure 3.4. The figure describes a “preparepush”operation, where agent X prepares itself to move an object Y, if the given conditions are satisfied in some state of the world.

The formalism can be interpreted as describing a whole class of actions which can be obtained by substituting for the variables X, Y, P – a schema.

Practical implementation of the situation calculus formalism in the PDDL and STRIPS notations is the notion of an *operator*. An operator is a data structure, which consists of several components:

- Name of the operator
- List of formal arguments
- Precondition expression

$$\begin{aligned}&\mathit{place}(P) \land \mathit{at}(X, P, s) \land \mathit{at}(Y, P, s)\\ &\qquad \land \mathit{agent}(X) \land \mathit{object}(Y) \quad \Rightarrow \mathit{Poss}(\mathit{preparepush}(X, Y), s)\end{aligned}$$

$$\mathit{Poss}(\mathit{preparepush}(X, Y), s) \Rightarrow \mathit{pushing}(X, Y, \mathit{Result}(\mathit{preparepush}(X, Y), s))$$

$$\begin{aligned}&\mathit{Poss}(\mathit{preparepush}(X, Y), s) \Rightarrow\\ &\quad (\mathit{pushing}(X, Y, \mathit{Result}(a, s)) \Leftrightarrow a = \mathit{preparepush}(X, Y)\\ &\qquad \lor (\mathit{pushing}(X, Y, s) \land a \neq \mathit{preparepush}(X, Y)))\end{aligned}$$

Figure 3.4: Situation calculus description for `preparepush` action using the possibility, effects and successor-state axioms.

- The effects expression

Some older systems split the effects into two clauses – "add" and "del" effects. The names stem from the usage – "add" effects are added to the resulting state after the action was performed (becoming true) and the "del" effects are removed, making them false. Because of the closed world assumption, this split is not needed anymore – to signify a "del" effect, it is possible to declare it as negation. Predicates which are false are simply not stored, achieving the same result as the "del" effects clause.

An example of such operator is shown in figure 3.5

```
(:action preparepush
  :params (?X ?Y)
  :precond (and (at ?X ?P)
                (at ?Y ?P)
                (agent ?X)
                (object ?Y)
                (place ?P))
  :effect (pushing ?X ?Y))
```

Figure 3.5: Operator version of the preparepush action

The name of the operator specifies the name of the action this operator represents. The formal arguments are substituted for when instantiating the operator. Such an instantiated (specialized) operator is called *action*.

The precondition expression declares when is the use of this operator (or precisely – its instances) admissible. The precondition expression has to evaluate to true, given some argument substitution (instance) and some state of the world.

Finally, the effects specify what will be the impact of performing the action on the current world state. There are many options available in PDDL 2.1, however the most useful ones are:

- Add-effects `((thirsty ?who))`
- Del-effects `((not (thirsty ?who))`
- Conditional effects
  `(when (thirsty ?who) (agitated ?who)),`
  this is equivalent to the construct:
  `if` `(thirsty ?who)` **`then`** `(agitated ?who))`
- Effects on numeric fluents
  `((when (<= (energy ?robot) 50) (recharge-needed ?robot)),`
  or `(decrease (energy ?robot) 10))`

The full syntax can be found in the specification [69].

3.2 VR-specific challenges

There are several challenges that are specific to the virtual reality systems and cannot be easily handled using only the generic data representation. They stem from the intrinsic properties of a virtual reality simulation system and how it is usually built.

3.2.1 Gap between geometry and semantic information

Virtual reality systems are usually created as computer graphics applications, with minimal consideration given to the semantic information. The typical data entity encountered there is a triangle mesh representing some geometric shape. There are also non-visual systems, using only haptics or audio, but these are special cases of limited interest in the context of the presented work.

These low level data entities are usually supplemented by a parallel set of structures containing high level information, such as object properties represented as predicates or something similar. Such a design is very typical, the high level information is only loosely coupled to the underlying geometric information for various reasons – usually efficiency of storage and retrieval of the information play major role.

Such a design frequently leads to consistency problems – the weak link between the low level geometric data and high level symbolic representation does not really allow to represent an object in the virtual world as a single whole. Several data structures have to be modified whenever the state of the object changes.

To illustrate the issue, let us consider a real-world car. Whenever the car moves, the amount of fuel available in the tank is reduced, the temperature of the engine grows and the location of the car changes at the same time. All of this happens "automatically", because it is driven by the elementary physical principles based on which the car works.

In comparison, let us assume that there is a car in the virtual environment. Typically, it would be represented as a geometry (e.g. a mesh or set of meshes) and an assortment of symbolic data. However, when the virtual car is moved, the simulator has to update all the data by itself. Even worse, the geometry can be affected (deformed, moved, etc.) by an external influence (user, physics engine etc.) without notifying the car simulator of the change. Suddenly the geometry of the object (what the user sees) is in a different state than the symbolic representation (model) of the car, they are out of sync.

This problem illustrates the need for a "bridge" between the low and high level information, which will tie these representations of the object together. One such mechanism will be described in the section 6.1.2.

3.2.2 Animation and its semantics

Actions of the virtual characters and objects are usually visualized in the form of animation, which is most often defined in the low level terms of key frames and time series of positions and orientations. This representation is sufficient to replay the animation (to perform the action), however to be able to reason about its consequences, more information is necessary.

In the section 3.1.4, representation of the action semantics was introduced. This approach can be very well used to describe the properties of an arbitrary animation which is changing the state of the simulated world, not only of the actions performed by the agents themselves.

Unfortunately, no action description is perfect. Typically, when performing animations in the virtual world, there can be side effects which are not taken into account in the semantic description of the action. For example, the moving car can crash into a fence and topple it over. In general, such events are impossible to take care of in the semantic description of the action because it would become too unwieldy and complex. However, the inhabitants of the virtual environment have to take the changed situation into account somehow, otherwise the symbolic representation of the world and the real state of it would be out of sync, possibly leading to incorrect behavior as described in the previous section.

This problem is fundamentally intractable unless every possible side effect is accounted for, which is impractical. Moreover, the agents can expect to know only about side effects of actions initiated by themselves, but not by other agents or users.

A partial mitigation of the problem can be the introduction of the sensing capabilities for the agents (to "discover" that the side effect occurred and the agent's beliefs are out of date) and robust recovery from the failed actions. If the action failed because of a side effect of another action "breaking" something, the agent has to be able to recover from the situation and potentially update its strategy how to achieve the goal (e.g. by re-planning).

3.2.3 Real-time response

The final challenge of the virtual reality applications is the problem of interactivity and real-time response. VR applications are usually designed as real-time simulators, where an immediate response to the stimuli given by the user is required.

This requirement has an impact on the world representation as well. Easy-to-use, simple but expressive systems have the advantage. The list-based prefix representation employed by the PDDL standard is a good candidate for this, because fast parsers and interpreters are readily available, with low overhead.

Another impact of the real-time constraint is on the design and implementation of the reasoners, which have to generate answers in reasonable time. This disqualifies a lot of AI research projects – their products are simply too complex/slow. Fortunately, there are planners available for the PDDL representation, which are fast enough for the envisioned application – collaboration with autonomous agents having moderate amounts of beliefs ($\approx$ few hundreds) and limited amount of possible actions ($\approx 10 \dots 50$). In particular, modern forward-chaining planners such Hoffmann's FastForward and Metric-FastForward [42, 41] can satisfy this requirement.

3.3 Human-agent communication

Human – agent communication assumes that there is some sort of "common language" between the two communicating parties. It has to be neither too low level, because that is too cumbersome for the human to use nor too high level, because it is difficult to process by the machine. Some intermediate form is necessary.

A suitable solution for the "common language" can be based on the formulas of the predicate calculus. It is neither too low level and it is high level enough for the humans to use directly if needed. The communication is achieved by sending messages composed of various predicate calculus expressions representing tasks and state of the world.

Predicate calculus is high level enough to be used as a base for various *translators*, which can translate the tasks and requests from the user(s) into something machine-comprehensible. Several examples of such *translators* will be shown in the next chapters:

- Translation from the natural language
- Graphical user interface allowing dynamic introspection and symbolic interaction with the system
- Random "story" generator to create a meaningful behavior in the scenario

All these examples have one feature in common – they use a common communication channel based on predicate calculus to communicate with the agents. To provide an easier interface for the user, the symbolic representation is translated from/into a different form – such as a natural language or graphical symbols. The high level task description allows to achieve this easily.

3.4 Summary

This chapter discussed the high level representation of the virtual world suitable for automated reasoning. The main contribution is the addressing of the problem of the task specification using both imperative and declarative definition.

The second topic of discussion was the introduction of the explicit state representation and action semantics using the predicate and situation calculus and numeric fluents. The actions are represented symbolically by mapping them to the STRIPS-like operators.

Chapter 4

Model of Collaboration

collaborate

1. *To work together, especially in a joint intellectual effort.*
2. *To cooperate treasonably, as with an enemy occupation force in one's country.*

The American Heritage®Dictionary of the English Language, Fourth Edition Copyright ©2000 by Houghton Mifflin Company.

The quote presents two meanings of the word "collaboration", however for the purposes of this thesis, only the first meaning is relevant. Collaboration as a joint effort between the machine and human is the main topic of this document.

In the following sections, the focus will be on both human-agent and agent-agent collaboration. The contributions of the delegation-based model will be described, together with the facilitator-based architecture used to implement it.

In the second part of this chapter the notion of teams and teamwork will be discussed. A team-forming protocol derived from the ContractNet protocol will be introduced and roles of the team-members will be discussed.

4.1 Agent-agent collaboration

Collaborative agents are a very large topic with many theories and technologies being developed and used. Some of these theories and the corresponding technologies were presented in the chapter 2.

The work presented in this thesis was heavily influenced by the work of Martin et. al. on the Open Agent Architecture (OAA), presented in [68, 14]. Open Agent Architecture has its roots in the older, blackboard-based systems but introduces new elements: *facilitator*, *delegation* and *ICL* – the inter-agent communication language. Martin positions his framework in [68] somewhere in-between the traditional agent-based systems built around KQML (see [26]) on one hand and heavily structured multi-agent systems using approaches such as BDI on the other hand.

According to Martin, the KQML-based systems leave too much structure implicit or unspecified (KQML is primarily a knowledge exchange language, not a programming framework), on the other hand systems using approaches like BDI impose too much specificity on the system developer. The structure of the system has to be known to the agents in a lot of detail in order for them to be able to cooperate in a meaningful way. Martin addressed this issue by the introduction of an information broker – the facilitator.

From the point of view of this thesis, the idea of facilitator-based “delegated computing” has several advantages compared to the other frameworks, such as DECAF or RETSINA. First and foremost, the agents are kept relatively simple, there is no need to provide explicitly any facilities for collaboration except the basic communication mechanism for dealing with the facilitator – a mean to submit requests to the facilitator and a queue for incoming replies. No coordination framework (e.g. for bidding for tasks or to deal with various location and matchmaking agents) is needed for agents providing basic services, such as access to various data stores or path planning.

Another advantage of facilitator-based system is the possibility of collaboration by broadcasting – often it is not necessary to enter into full fledged collaboration with an agent but to broadcast the request for help instead and expect somebody to deal with it. This interaction is typical for requesting services such as path planning between two points (the agent has no reason to care who performs the task) or asking somebody to open the door in the virtual environment (similar to calling for help without addressing anybody specifically). In the facilitator-based system, such feature is easily implemented by delegating the task to the facilitator which will delegate the task to all suitable agents having required capabilities.

The Open Agent Architecture as developed by Martin doesn’t have an explicit notion of teamwork or collaboration between the agents. The agents are collaborating by virtue of delegating specific sub-jobs to the specialized agents, however there is no notion of a team. In fact, as will be described in the next section, the agent does not even need to be aware of the identity of the agent it is collaborating with, which has profound effects on the design and implementation of the agents.

In the extreme case, the collaboration can be purely emergent, each participating agent is simply processing input given to it and returning results to the originator of the request without an explicit intent to collaborate. Such model is very close to the model seen in the blackboard architectures, where agents are watching the blackboard for the tasks assigned to them and writing back the results of the processing of these tasks, without having any notion of the collaboration or even awareness of the surrounding/cooperating agents (“broadcasting”, see above).

The technology developed by Martin in his OAA framework is useful for solving many practical problems, however it is not sufficient if we consider virtual reality simulation where the collaborating agents are represented by virtual humans. Application of the OAA principles to virtual humans could lead to unrealistic behavior, such as agent A asks agent B to open a door for him (e.g. because agent A is encumbered by something), B complies and after passing through the door, the agent A asks *another* agent C to close that door again. This is not what human would do – most likely he would ask the same agent/human to close the door again, not a different one (which may be far away, for example).

The reason for such unrealistic behavior is that OAA-style of collaboration is ultimately “stateless” – the framework has no notion of explicit collaboration and the agent doesn’t remember (or even know as explained above) who its collaborators are. Such approach works well for “one-off” tasks, where the requesting agent does not need to the collaborating agent for anything else later.

Unfortunately, unless we restrict ourselves to very simplistic cases, even simple collaboration in a multi-agent system requires some notion of a *team* and its state. Let us consider for example a police unit trying to protect a building from a crowd of rioters. The unit has a hierarchy with a commander – *team leader* and the subordinate *team members* – individual policemen. If there was not any notion of a team, any meaningful cooperation would be impossible, because the commander has to know where and what every member of its team is doing. With the standard OAA approach, he may not even know who the members of his unit are!

One of the innovations of the presented work is the extension of the OAA concepts with the explicit notion of teamwork and collaboration. To achieve this goal, OAA is extended by using a ContractNet protocol to form explicit teams, while keeping the other advantages of OAA.

4.2 Delegation

In the Open Agent Architecture, the central role is played by the process known as *delegation*. The basic idea of the delegation process is to commit another agent to the given task, typically the task is delegated to a specialized agent able to solve it.

In OAA, Martin introduced a special form of delegation – delegation via the *facilitator* (a special agent acting as a broker). The main reason why the facilitator was introduced is that it allows the system to act as an all knowing "oracle" – each agent can delegate the tasks to be solved to the facilitator without having to know the identities of the agents which are going to solve the task. Figure 4.1 shows a typical architecture of a OAA-based application.

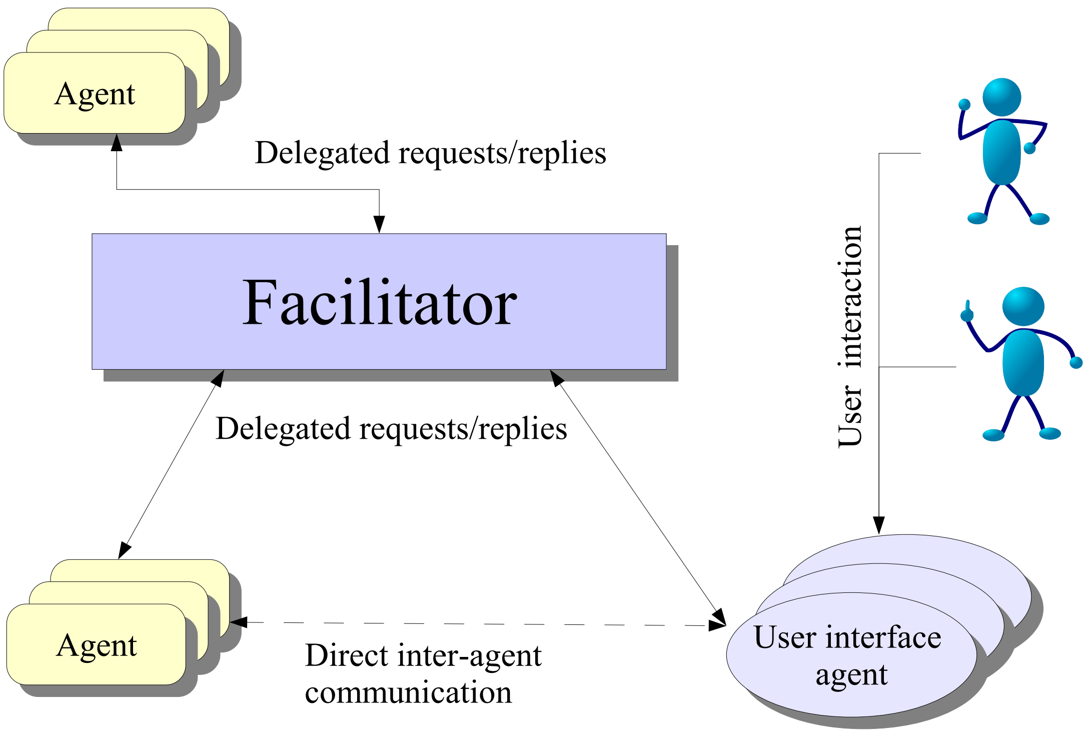

Agent
Delegated requests/replies
Facilitator
User interaction
Delegated requests/replies
Agent
Direct inter-agent communication
User interface agent

Figure 4.1: Diagram of an Open Agent Architecture system

The figure shows three main parts. There is the facilitator, the agents providing services and finally the agents providing user interface. All of these agents are supposed to communicate with one another mainly via the services of the facilitator, to avoid the necessity of having to "hardwire" the identities of the cooperating agents. However, the direct communication between the agents is not forbidden, it is still possible for the agents to talk directly to each other without the use of the facilitator's services.

4.2.1 Role of the facilitator in OAA-like system

The facilitator plays a central role in any OAA-derived system. It fulfills two main goals:

- Central data storage (similar to a blackboard)
- Matching the incoming requests to the capabilities of the agents and delegating the tasks to the agents for execution.

The agents communicate with the facilitator by the means of *solvables*. Solvables consist of three main parts – the goal (functor), the parameters and the permissions. The goal specifies what has to be done. The parameters specify additional information (e.g. a destination for moving or an e-mail address for sending a report). Finally, the permissions specify restrictions placed by an agent on how other agents can use its services – for example restricting access only to certain agents or only to certain mode of usage, such as read-only.

There are two types of solvables available:

- Data solvables
- Procedural solvables

Data solvables represent pieces of information stored in the facilitator and retrievable and modifiable by the agents at runtime. Declaring a data solvable to the facilitator amounts to writing a piece of data into a shared blackboard. Essentially, it is possible to emulate a blackboard-based system using the data solvables.

Procedural solvables represent *capabilities* advertised by the agents, such as the ability to manipulate object, the ability to walk, ability to block streets, etc. In essence, each agent declares the actions it is able to perform, together with the required arguments and potentially additional information to the facilitator. The capabilities are used to resolve the delegation requests from the other agents.

Martin used a special ICL[1] language based on Prolog for this purpose. For example, a simple e-mail agent may provide two capabilities (interface) and one data solvable shown in fig. 4.2, in ICL syntax.

The solvables are declared using three components, as described above. First is the goal (e.g. `send_message`), then arguments, where +, - indicate input and output parameters respectively. Finally, there is indication of type of the solvable (procedural or data), the callback function to call in case of a procedural solvable and permissions.

The fundamental interface to the OAA facilitator is the `oaa_Solve` procedure. The argument to this procedure is an *event* specifying the goal and the corresponding parameters. The second optional argument may contain a set of *constraints*, specifying certain meta information for the facilitator – such as that the calling agent is interesting only in a single solution or that the solutions should be restricted only to certain agents. The result of calling this procedure is composed out of the results returned by the agents the task was delegated to.

An important feature of the OAA facilitator is the ability to *decompose* compound tasks into simpler ones and to delegate the sub-tasks to different agents. For example, figure 4.3 illustrates how a request "Fax it to Bill Smith's manager" can be translated into ICL and decomposed.

[1]Interagent Communication Language

```
solvable(send_message(email, +ToPerson, +Params),
[type(procedure), callback(send_mail)],
[])

solvable(last_message(email, -MessageId),
[type(data), single_value(true)],
[write(true)]),

solvable(get_message(email, +MessageId, -Msg),
[type(procedure), callback(get_mail)],
[])
```

Figure 4.2: Capabilities of an e-mail agent

```
oaa_Solve((manager('Bill Smith', M), fax(it,M,[])),
          [strategy(action)])
```

Figure 4.3: Compound goal being decomposed by the facilitator

The stanza `manager('Bill Smith', M)` specifies that a Bill Smith's manager has to be retrieved and substituted for variable `M`. The second part specifies, that `it` has to be faxed to the person identified by the content of `M`. Both tasks, finding Bill's manager and faxing the document may be handled by different agents and the facilitator handles the decomposition and re-assembling of the request transparently. In case of the original OAA, the facilitator was implemented using the built-in features of Prolog, in this case mainly *unification*. How a similar decomposition process can be implemented will be described in detail in the section 6.2.

4.2.2 Facilitator design

The original Open Agent Architecture facilitator was developed at SRI in Prolog with bindings to a few other languages. The design of the ICL language shows this heritage clearly. In order to provide a better match for the world representation described in the previous chapter, a new facilitator was implemented by the author of this thesis which simplifies certain aspects of the original design.

```
(at gino frontyard)     ; data solvable
(move gino ?from ?to)   ; procedural solvable

; compound task solvable from the previous section
(and (manager 'Bill Smith' ?m) (fax ?a it ?m))
```

Figure 4.4: Predicate calculus solvables

First of all, the original ICL language was replaced by the predicates of predicate calculus written in the prefix form, as introduced in the chapter 3. The notion of data and procedural solvables was retained, however the solvables are now represented by predicate expressions denoting semantic relationships between entities. The figure 4.4 illustrates how such solvables can be specified.

The differences in between the original OAA facilitator and the proposed facilitator design can be summarized as follows:

- No permissions.
- No callbacks or handlers.
- The facilitator does not assemble the results to be returned to the delegating agent.

The word "permissions" implies some security mechanism, however in the original OAA design it was used also to simply specify, whether the solvable is writable or read-only. In the presented design the solvables can be only declared or undeclared (deleted). There is no value to change, if e.g. a value expressed by some solvable changes, the old solvable has to be removed and new one declared. Such design simplifies the implementation of the facilitator (no need to track the value changes) without impact on the functionality.

For the purposes of this thesis the issue of security and access restrictions was not considered. Of course, in case of industrial deployment of such system, the security layer will have to be added.

The original OAA facilitator by Martin requires that for each procedural solvable a *handler* is defined, i.e. a function that has to be called to process the subtask (such as retrieve a value from a database) and return a partial result. This design was deemed unsuitable for the purpose of this thesis because it doesn't play very well with the predicate calculus expressions used to express solvables and delegated tasks. Predicates can evaluate to either true or false, however a function can return an arbitrary value, such as number or a string – such expression could be used only in special cases, such as as a predicate argument and care would have to be taken to reject invalid expressions (e.g. `(and (3 (at gino house)))`). Such requirement would complicate the processing a lot.

To be able to remove the handlers, it is necessary to find a replacement solution such that the delegation ability of the facilitator is retained. We propose to achieve this goal by restricting the procedural solvables to be always in the form `(functor agent arg1 arg2 arg3 ...)`. The functor specifies the action to be performed, agent is the identification (name) of an agent which declared this capability (solvable). It is possible for multiple agents to define the same action/capability, however they will always differ at least in the name of the declaring agent – "owner" of the solvable. The request dispatch/delegation will be then performed simply by matching the incoming requests against the table of declared solvables. The matched solvables/capabilities directly determine which agent is to receive the delegated task. This is a service commonly known as *yellow pages* or match-making. The implementation details of this process will be described in the chapter 6.2.

In the original design of Martin, whenever the agent delegated a task to the facilitator for solution, the facilitator was responsible for delegation of the possible sub-tasks to the collaborating agents, then collecting the results and assembling the final result form which was returned to the originator of the request. Moreover, the process could have been influenced by one or more *strategies* which are used to further constrain the solution – e.g. if three agents return differing results for a query and the originator requested only a single result, the strategy specifies how the final result should be selected.

Such approach leads to very complex processing because the facilitator has to track each request, all its sub-requests which were delegated to other agents for processing and also all the replies to these sub-requests. In a distributed multi-agent system with many agents this is a non-trivial task without becoming a bottleneck. Finally, the facilitator has to support complex strategies, because the originator of the request has little influence on how the final result is composed. The only way how it could influence this process is by specifying the strategy when delegating the task to the facilitator for processing.

To address this complexity we propose an alternative solution. If we observe that the main complexity of the facilitation task stems from having to track and recombine the solution requests and the replies to them, there is a natural response possible – to charge the originator of the delegated task with the job of processing the replies. We can achieve this easily by attaching a "return address" in the form of an unique identifier of the delegating agent to each request. During the delegation of the sub-tasks to be solved the facilitator delegates this "return address" together with the tasks to be solved to the agents and the agents send their results directly to the source of the original task. The facilitator does not have to keep track of the replies to the requests (potentially to many requests from multiple agents at the same time) and does not become a bottleneck easily. The throughput of the facilitator will be limited only by the speed of the matching process and the available network bandwidth.

The schematic diagram 4.5 illustrates this process in case of a task delegated to the facilitator by the user interface agent (for example a natural language translator or some sort of a GUI). The behavior of the system would be exactly the same if it was a regular (non-UI) agent delegating the task.

This approach has two advantages and one drawback. The first advantage is that the facilitator does not have to be concerned with the tracking of the sub-tasks and their results. It simply determines which agents are to be involved in the solution of the particular task and delegates the job to them. The facilitator's reply to the originator of the task is only information whether the task was successfully delegated for solution or not and how many replies are to be expected. In case of a data solvable query, the facilitator returns the solution directly.

The second advantage is that the agent which originated the request can directly deal with the results received and apply whichever post-processing algorithm it desires. The complex strategies used by original OAA facilitator can be replaced for the most part by the combination of this processing on the originator's side and simple constraints on the facilitator's side (e.g. "return only single result" or "return N results"). It is assumed that most agents will not need the full complexity of the original OAA strategies to select from the received results and will implement only the ones required depending on the role the agent is playing in the system. However, the original OAA facilitator has to support all of them in case one agent requests the strategy.

The drawback mentioned is the partial loss of the task decomposition capability which the original OAA facilitator has. For example, if the example from figure 4.3 is considered, the task of retrieving the manager of Mr. Smith cannot be meaningfully implemented by an agent and delegated to it. The reason for this is that the agent which originated the request has no means how to meaningfully determine how the request was decomposed by the facilitator. It would receive a list of Mr. Smith's managers from agents able to resolve the `manager()` goal and the results of faxing something to them from agents able to resolve the `fax()` goal, however without any information how to combine these values into a meaningful result.

Fortunately, this functionality loss is not a major issue. First, in cases similar to the fig. 4.3 example, the `manager()` part of the compound goal can be easily handled by a data solvable which is then resolved by the facilitator directly (similar to a database look-up), without the need for delegation to another agent. Compound goals involving independent actions to be performed by multiple agents are still handled without problem – such as moving two agents to the same place and asking another agent to close the door.

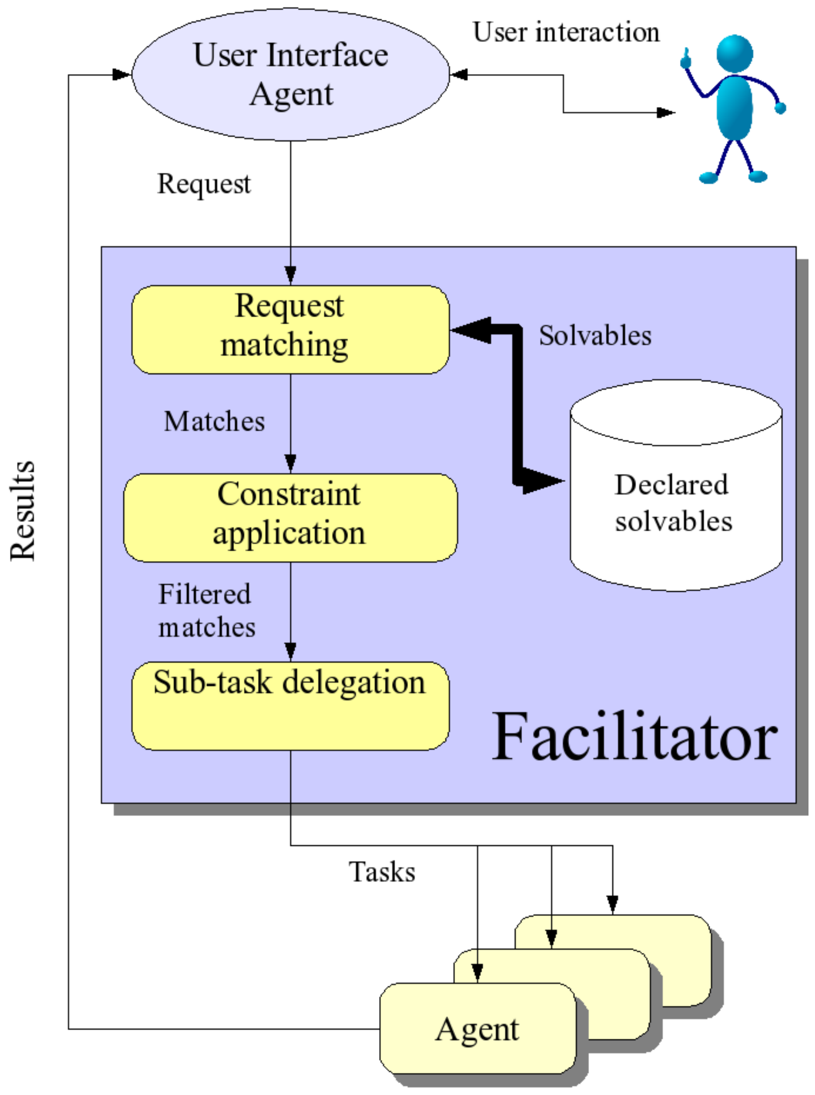


Figure 4.5: Request processing in the simplified OAA system

Second, if a true task decomposition is required for solving more complex problems, a more powerful mechanism can be deployed – an action planner. In such case the originating agent will receive a *plan* specifying how to solve the declared goal, potentially including collaboration with other agents and/or additional information retrieval. A collaborative problem solving approach based on this mechanism will be described in detail in the chapter 5.

4.2.3 Matching requests with offered services

In order for the facilitator to be able to fulfill its delegation and compound goal decomposition role, it has to be able to match incoming requests with the offered (and presumed available) services.

As described in the previous sections, the agents declare their capabilities in the form of procedural solvables and their state in the form of data solvables which in turn have the form of predicate calculus predicates. The declared predicates are presumed to hold (to evaluate to *true*), as described in chapter 3 .

To match and decompose the incoming request, it is necessary to have an algorithm/function which can determine whether two expressions are matching. The unification-based technique presented in this section is state of the art technique well known from literature but adapted to the specific requirements of the proposed facilitator implementation.

The algorithm commonly used to perform the matching is known as *unification*. The basic idea of unification is that the two expressions are recursively compared. If the expressions contain variables then such substitution for them is sought that makes the two expressions equal.

The unification algorithm takes two expressions (sentences) and returns an *unifier* for them if one exists:

$$UNIFY(p,q) = \theta \; where \; Subst(\theta,p) = Subst(\theta,q)$$

$$Subst(\theta,z) = result \; of \; applying \; substitution \; \theta \; to \; expression \; z$$

The unifier placing less restrictions on the variables in the resulting substitution is called *more general*. For every unifiable pair of expressions there is exactly one *most general unifier* that is unique up to renaming of the variables. The algorithm to compute the most general unifier is well known and is available in literature, for example in [64, 5, 91].

The problem with the algorithm to compute the most general unifier is its complexity – it is quadratic with regards to the lengths of the terms being matched because of the *occurs check*. The occurs check is required to verify whether the variable being matched against a term occurs itself inside the term. If it does, it is not possible to construct a consistent unifier. This has negative impact on the performance of the facilitator in case that complex expressions with many terms have to be matched.

To demonstrate the unification process for matching an incoming request in the facilitator on a practical example, let us consider the expressions in the figure 4.6:

```
(at gino door)         ; declared state (data solvable)
(move gino ?from ?to)  ; declared capability
(move carlo ?from ?to) ; declared capability

(move ?a ?p table)     ; agent request

; compound request
(and (at gino ?place) (move carlo ?somewhere ?place))
```

Figure 4.6: Unification example

The first three expressions are typical examples of declared solvables which the facilitator may know about – symbolic position of an agent and capabilities two agents. Let's assume that the request (imperative task) `(move ?a ?p table)` was submitted to the facilitator for processing.

To determine what has to be done the expression is unified with all known solvables. The data solvable `(at gino door)` can be rejected as non-matching outright, because it has different length[2]. For the second declared solvable it is possible to observe that if we substitute the variable `?a` with `gino`, `?from` with `?p` and `?to` with `table`, the expressions would be equal. This substitution is customarily written as $\{gino/?a, ?p/?from, table/?to\}$. The notation X/Y means that X is substituted for the variable Y. This substitution of the variables is also commonly referred to as *binding*.

There may be multiple unifications possible. For example the following substitution:

$$\{carlo/?a, ?p/?from, table/?to\}$$

leads to a valid match as well, but for a different agent. It means, in this case, that two different agents have the requested capability and both may be asked to realize the task, depending on the possible further constraints the requesting agent specified.

The last example of a compound goal is more complex. To resolve the compound goal the facilitator has to recursively process each sub-expression of the original goal and combine the results. Let's ignore the logical connective for the moment and try to match only the sub-expressions (or in other words to establish whether they are known to be true in the world given by the solvables known to the facilitator). The first sub-expression `(at gino ?place)` can be obviously matched only against `(at gino door)` with substitution $\{door/?place\}$.

The second sub-expression `(move carlo ?somewhere ?place)` can obviously match only `(move carlo ?from ?to)`, with the variable substitution:

$$\{door/?place, ?somewhere/?from, door/?to\}$$

The substitution has to be consistent, once a variable is bound to an expression, every instance of that variable in the whole compound expression has to be substituted with the same expression – in our case the first sub-expression established the binding $\{door/?place\}$ and therefore the second sub-expression cannot bind its instance of $?place$ to anything else than $door$ to stay consistent.

The consistency requirement across the compound expressions allows to express complex task specifications. The last example is essentially equivalent to saying "Carlo, move from wherever you are to the place where Gino is."

The logical connectives which were ignored until now determine how the results of the sub-expression evaluation are combined. A common approach used to process the compound expressions is the stream or filter method, described in a well known book by Abelson et al. [5]. The discovered set of substitutions (unifications) is treated as a *stream* passing through a chain of filters formed by the logical connectives. At each node of the chain and for each substitution (frame), the matcher goes over the known solvables and either extends the frame with new variable bindings or indicates that a match failed.

The frame is then further processed depending on which logical connective is represented by the chain node. The proposed facilitator is able to process expressions containing the logical connectives `and`, `or`, `not` and to evaluate equality predicates (>, >=, =, <, <=).

- `and` – logical conjunction. The frame passes through a node for each sub-expression in series. It must generate a match at each step, otherwise it is discarded.
- `or` – logical disjunction. The substitution has to pass through sub-expression nodes in parallel and at least one has to generate a match, otherwise the frame is discarded.

[2]When interpreted as a length of a list in LISP sense.

- `not` – negation. The frame must not generate a match, otherwise it is removed from the stream.
- Equality predicates – the predicate is evaluated, if it evaluates to false, the frame is discarded.

After the whole expression is evaluated, the stream of substitutions is inspected. If it is empty, an error is reported to the calling agent because there is no way how to resolve its request. If there are some matches, then each of them is processed in turn and delegated to the other agents for final resolution (for the procedural solvables) or simply returned to the caller (in case of data solvables).

The implementation details of a facilitator using the described expression evaluation process will be described in section 6.2.

4.2.4 Global world state

In order for the facilitator to be able to answer queries about data solvables and to delegate the requests to the agents it has to have the required information in the first place. It is achieved by having the agents declare (upload) all necessary information about their capabilities and their state in the form of the solvables represented as predicates, as described in the chapter 3.

The information contained in the facilitator defines a global state of the simulated world and the agents are expected to update it whenever their status changes. In this way the facilitator plays also a role of a global blackboard which each agent or user wishing to interact with the system can easily query for information to complement their own beliefs – such as enumerate all agents, find out where an agent is and what is its state or what are the properties of some objects.

The main disadvantage of using the facilitator in this way as a combined broker/data storage system is that a very careful consideration has to be given to the amount of data being kept there. Since the processing of the incoming requests requires that the expressions of the request are matched against every declared solvable[3], the time to resolve each request will grow with the size of the database. The increase is not linear, because complex expressions may require multiple unification passes over the database. Martin et. al. in their work on Open Agent Architecture acknowledge the possibility of the facilitator becoming a bottleneck for the system and propose a solution using a multi-facilitator setup, where multiple facilitators are used to satisfy the requests of the clients.

4.3 Teamwork

The facilitator enables the agents to delegate tasks to another agent. This mechanism forms a basis of a collaboration scheme, where the agents can utilize the capabilities of specialized agents to perform complex tasks. Unfortunately, as described before, it is not sufficient for practical purposes to have only pure delegation-based collaboration. The inherent statelessness of this approach causes problems in simulations where it is important to know the identities of the collaborating agents and to coordinate a team of agents.

To supplement the facilitator-based system and to allow the teamwork as usually understood by humans – i.e. a team of several members committed to certain task is formed and its leader is coordinating the efforts of the team members towards the desired goal, a new team facility has to be provided.

[3]Some optimizations are of course possible, e.g. pre-filtering solvables based on their functors.

4.3.1 Forming teams

It could be considered that the agents collaborating by the means of delegation through the facilitator are a team of cooperating agents. However, this is not the case, because the agents do not even have to be aware of each-others existence by default. To form a real team the decision to do so has to be a conscious effort (intention) of the agent.

Most cooperative systems do not address the issue *when* to form a team and neither *how* to form it. Usually the team is either defined *a priori* – for example in [73] or in [38] or it is expected that the team somehow exists and the issue does not have to be dealt with.

In cases where the work is concerned explicitly with the team creation, the described approaches focus on agents built on BDI frameworks (such as [93], or various joint intentions/commitments models described in the related work section). Such team building models were described by Shehory and Kraus in [55, 98].

These two cases represent two extremes – the first one is the situation where the teams are "hard-wired", simplifying many issues in the process. Unfortunately, such approach work only for specialized applications. The second extreme is the case of fully autonomous agents trying to fulfill their own goals. This case leads to complex negotiations and deliberation schemes needed to bring the agents into a single team and to align their motivations towards the common goal.

Neither of these two cases fits the OAA-derived model well. The main idea of OAA-based systems is the added level of indirection represented by the facilitator to avoid having to explicitly "hard-wire" the inter-agent relationships into the system. The first case, where the teams are pre-defined, is therefore excluded because it goes squarely against this idea.

The second case with a complex deliberation process does not fit the facilitator-based model very well neither. The deliberation processes are designed for fully equivalent agents which are acting on their own towards their own goals. The delegation model, where one agent delegates task to another one, implies a certain hierarchy (which can be created on the fly and be different each time), thus negating the needs for the complex deliberation. For example, since the delegating agent asks for a specific goal to be satisfied, there is no point in negotiating something like a joint intention to determine what has to be done by the team.

Somewhere in-between these two extremes are frameworks such as GPGP [16], RETSINA [99] and DECAF [36]. All these frameworks represent the tasks using the TAEMS [17] formalism. These frameworks provide dynamically created teams of agents which are formed on as-needed basis depending on the task being solved. However, the frameworks rely on the formal description of the tasks using the hierarchical TAEMS task structures which encode also the information about the required task decomposition, such as that in order to perform a task X the agent has to first acquire information a,b and c and task Y has to be performed.

The production of the TAEMS data is quite laborious for real-world applications with many possible tasks and highly domain specific. To minimize this problem it is desirable that the agent should be able to decompose the task autonomously and find the solution with minimal human input, given only the set of primitive actions it is able to perform. Therefore the decision was made to use the simple OAA-like model with facilitator-based task delegation and augment it using propositional planning.

To summarize the situation:

1. The team goal is explicitly given.

2. There is an implicit hierarchy in the team involved in solving the problem.

3. The team has to be created dynamically, not predefined.

4. The data to support reasoning should be easily obtainable and not laborious.

A solution tailored to fit these requirements can be devised using a standard protocol maintained by FIPA[4] – the Contract Net interaction protocol.

4.3.2 Contract Net and team forming

Contract Net belongs to the family of interaction protocols developed by FIPA in an effort to standardize the information flow and the inter-agent communication in multi-agent systems. Other popular standard maintained by FIPA is for example KIF (knowledge interchange format).

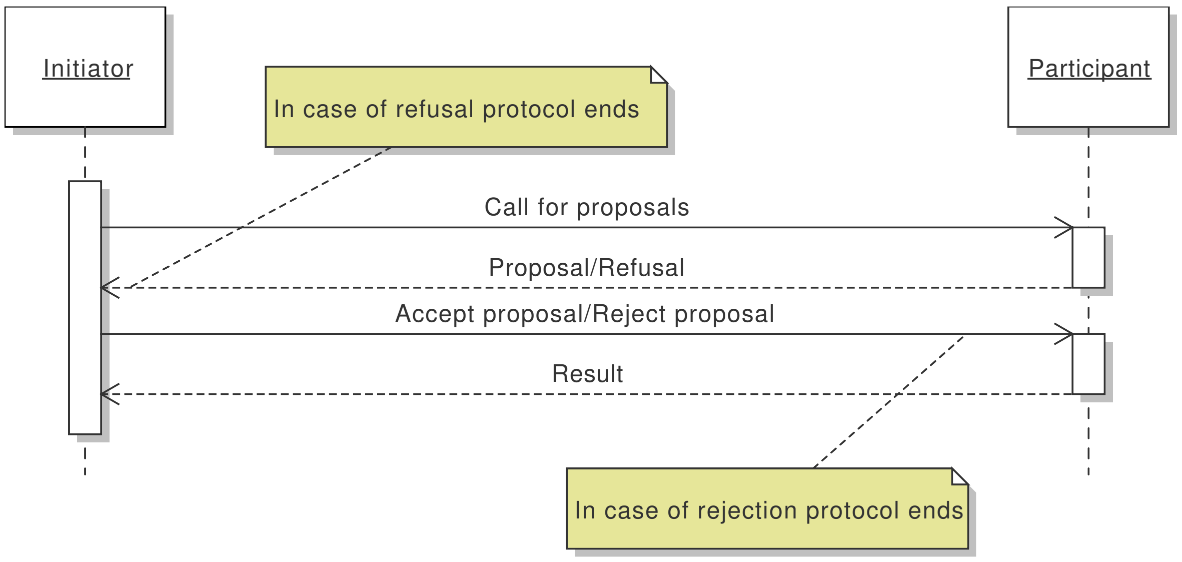

Initiator
In case of refusal protocol ends
Participant
Call for proposals
Proposal/Refusal
Accept proposal/Reject proposal
Result
In case of rejection protocol ends

Figure 4.7: Sequence diagram of the Contract Net protocol

Contract Net is basically a variation of a three-way handshake well known from computer science. Figure 4.7 illustrates the idea of the protocol. The initiating agent sends out a request for proposals soliciting offers for collaboration or for solution to the particular problem. The participating agents answer either with refusal if they are not interested in collaboration or with a solution proposal. The choice of the answer is up to the agent and is made depending on the internal beliefs and state of the agent (e.g. a member of one team will refuse to join another team).

The initiator can either accept the proposal or refuse it. There could be more "strings attached", such as response deadlines or different types of answers, but the basic idea is as described. The full specification of the protocol can be found in [29].

The three-way handshake presented by Contract Net lends itself well to the task of forming teams to solve complex goals. However, for that purpose it needs to be slightly modified. The modified scheme is in figure 4.8.

The team forming protocol works as follows:

1. The initiator (the future team leader) broadcasts request for help (a team forming proposal). This is done by delegating a `(join-as-teammate ?a)` goal to the facilitator.

[4]Foundations of Intelligent Physical Agents, http://www.fipa.org/

2. The prospective team members which are willing to join the team answer with the proposal to join team `(joining-team AGENT)`, making their identity available to the team leader.

3. The team leader selects the suitable team member(s) from the pool of candidates and asks them to commit to the team by delegating them solvable `(commit-to-team AGENT TEAMLEADER)` (acceptance of the proposal). The remaining agents are rejected by sending them the proposal rejection goal `(reject-teammate AGENT)`.

4. Finally, the team members confirm their commitment to the team.

The rationale for this three-way handshake is simple. The team-forming agent needs to have an option to pick suitable team members based on certain criteria – such as distance from itself or special capability requirements (even though capabilities are handled specially for the most cases). The special commitment step is needed to “lock” the agent to the team. Otherwise the agent would be available for other team leaders forming their own teams and that is rarely desired.

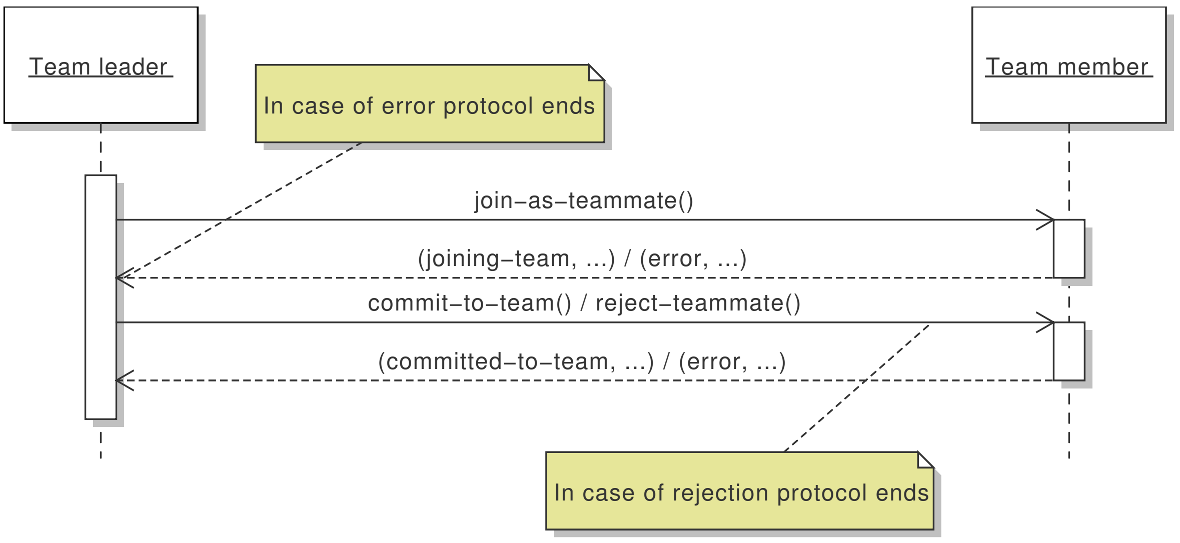

Team leader
In case of error protocol ends
Team member
join–as–teammate()
(joining–team, ...) / (error, ...)
commit–to–team() / reject–teammate()
(committed–to–team, ...) / (error, ...)
In case of rejection protocol ends

Figure 4.8: Sequence diagram of the team forming protocol

Finally, there is one more change which is not visible from the sequence diagram of the negotiation. It does not make sense to recruit team members which are unable to solve the problems which the team leader is going to delegate to them. To avoid this problem, the required capabilities are specified as solution constraints during the solicitation step. In this case the facilitator is able to filter out unsuitable candidates (which didn’t declare the required capabilities) and the `(join-as-teammate ...)` proposal is delegated only to agents which have reasonable chance to function in the team being created.

The only problem which remains to be addressed is the issue how to determine in advance which capabilities will be required. Usually, the required capabilities are calculated from a *plan* created by the team leader (utilizing an action planner). The plan consists of sequence of actions to be performed by the team. It is possible to extract actions for each participating agent, generating the minimal set of required capabilities necessary for the team member to have. This process will be analyzed in more detail in the chapter 5, because there are some specific issues to be resolved before an usable plan can be generated.

4.3.3 Roles in the team

There are two implicit roles in the teams created by the modified Contract Net protocol – the team leader role (the initiator of the team forming) and team member role (all other collaborating participants). The team is designed as hierarchical with two levels – the explicitly specified leader and the team members.

The role of the leader is a very important one – the leader coordinates all activity of the team by distributing work to be done to the team members. This collaboration scheme contrasts with schemes which are leader-less and the agents distribute the work among themselves by deliberation (if there is some kind of team plan – e.g. the multi-agent STEVE variant in [89] or [38]) or simply try to further the common goal by independent actions and observing and reacting to the actions of the others (as the simulated dinosaurs in [33] by Funge).

The hierarchical structure with explicitly established roles may seem as artificially limiting, however that is the structure which is probably the most common one in human society (manager $\leftrightarrow$ employee, military chain of command, etc.) and the most natural to simulate.

An added benefit of having a hierarchical structure with an explicitly defined leader is a simple fitting of such system to the agents collaborating by delegation – it naturally occurs that the delegating agent can act as a team leader and the other agents as team members. In the team established by the Contract Net-based process described above the role assignment (team leader vs team member) stays stable during the existence of the team, with the leader coordinating the team efforts.

It would be possible to allow exchange of roles (a regular team members becomes leader), however for practical purposes this is not necessary because the team can be always reformed. An agent can seek to become a team leader when it has a problem to solve which cannot be solved by the agent itself. Because in all likeliness this problem is substantially different from the problem the original team was formed for, the cleaner and more orthogonal solution (single team for a single purpose) is to form a new team even though it may contain the same agents as the old one and with new role assignment.

The only exception is the case when an existing team leader delegated a sub-task to the agent which is unable to solve it alone and requires a sub-team to be created. This issue was not addressed in the current framework and remains open for future research.

4.4 Human-agent collaboration

Human-agent collaboration is understood as a cooperative activity, where human user(s) is working in concert with the collection of (software) agents to solve a particular problem.

We have identified several partial problems in the human-agent collaboration which are addressed in the presented work:

- Establishment of the place and role of the user in the agent hierarchy.
- Communication of the task from the user to the agent or vice versa.
- Communication of the task results to the originating agent/user.
- Provision of human-oriented interface (e.g. by means of the *translators* mentioned in the previous chapter).

The user's place in the system can be a complex matter. There are multiple possibilities how the user can be involved in running of the simulation. For the purposes of this work three variants (interaction levels) were identified which represent three fundamentally different ways of interacting with a VR system. They are enumerated here in from the most low level one towards the most high level one.

1. Direct interaction with the VR environment. The user does everything alone (by direct manipulation, using tools or commands to the simulation system).

2. Team work with one or multiple collaborators. The team members can be either other human users or autonomous agents, either embodied or not. The user's activity is mixed, some tasks are performed directly, however other tasks are delegated imperatively to the other collaborating team members.

3. High level team work. The user only declaratively specifies the goal and shifts the burden of figuring out how to do it and the performance itself to the other agents/human users. These can potentially form collaborating teams to address the user's request.

Most of the existing VR systems implement only one of the first two interaction levels, for example [3], [54], [56], [88], and many others. The third mode is rare in virtual reality applications because managing automatic task decomposition and reasoning on high-level tasks delegated to the machine for processing is a non-trivial problem for people without background in artificial intelligence. Author is not aware of a VR application enabling the user to delegate whole problem solving tasks to the teams of autonomous agents, at most it is possible to delegate single orders, such as "walk to place X" or "perform task Y".

Moreover, most applications implement only exactly one of these levels. Very rarely is it possible for the user to choose the method he wants to use to solve a particular problem, which is an important problem to address. There is no "one size fits all" solution. Giving the user the choice how to approach a problem and permits to utilize better the user's creativity. From a technical standpoint the advantage is also clear – each interaction level/mode acts also as a sort of backup for every other. If some mode is unusable or impractical for whatever reason it may still be possible to use the others. Typical issues which render some mode unusable are lack of precision in hardware used for interaction (such as trackers, gloves, haptic hardware), it would take too long/too much effort to complete a task by direct interaction (e.g. complex manipulation tasks without proper tactile feedback) or it is impractical (e.g. positioning guards in strategic places by having to tell every single one where to go and what to do).

It is important to note that there is no "mode switching" in the application. All three interaction levels have to be available all the time at the same time in order to be truly useful to the user. Let's imagine that a trainee in a police training system has a task to secure a perimeter around a certain building with his squad. He has these options available:

- Direct interaction. The trainee can directly control each member of the simulated police squad and station him to the correct place at the perimeter.

- Teamwork mode. The trainee coordinates the squad members by issuing them orders to get them placed into the proper places.

- High level mode. The trainee orders the squad commander to secure the designated building. Everything else is automated – the squad commander figures out whom to send where and how exactly is the goal to be fulfilled.

In both the teamwork mode and the high level mode the user has the option to intervene, either by issuing a direct command or by direct interaction, regardless of the "main" level he chose to solve the problem.

The direct interaction level is the lowest control level reasonably possible in a VR system. The user is allowed to directly control the virtual characters (to "drive" them using some kind of input device, such as keyboard, mouse, joystick, etc.) and to directly examine and manipulate objects present in the VR system. Such direct control poses special challenges on the symbolic world representation because the direct manipulations are not tracked by corresponding changes to the world state. This introduces inconsistencies with which the agents and other users have to be able to deal with. Typical consequence of such an inconsistency is an action plan failure because the agent used out-of-date information to build the plan. These consistency problems can be mitigated by enabling the agents to better perceive their environment, for example by frequent sensing or by direct notification of the changes by the system.

In order to be able to take control of the virtual characters and object in the virtual environment a new "ghost and puppet" framework was designed by the author. It allows sharing and/or switching the control of the virtual character or object with an autonomous agent. Alternatively, it is possible to use multiple control mechanisms (such as a keyboard + mouse combination vs. a motion tracker) to control the same entity in the virtual world. The implementation details of this framework are described in the section 6.1.2.

The teamwork mechanisms needed for the other two interaction levels can be easily adapted from the collaboration framework for the autonomous agents, as described in the section 4.2. We propose to use the task delegation as the main collaboration mechanism, as used by the framework. This maps very well to the typical usage scenarios for the human users – the user either delegates some parts of the task assigned to him/her to the team members or alternatively receives some work to be performed. Examples of such interactions are human asking for information from an autonomous agent (e.g. virtual receptionists Karen in [77]) or autonomous agent looking for advice from the human (e.g. [40]) or asking him to perform some task (e.g. [90]).

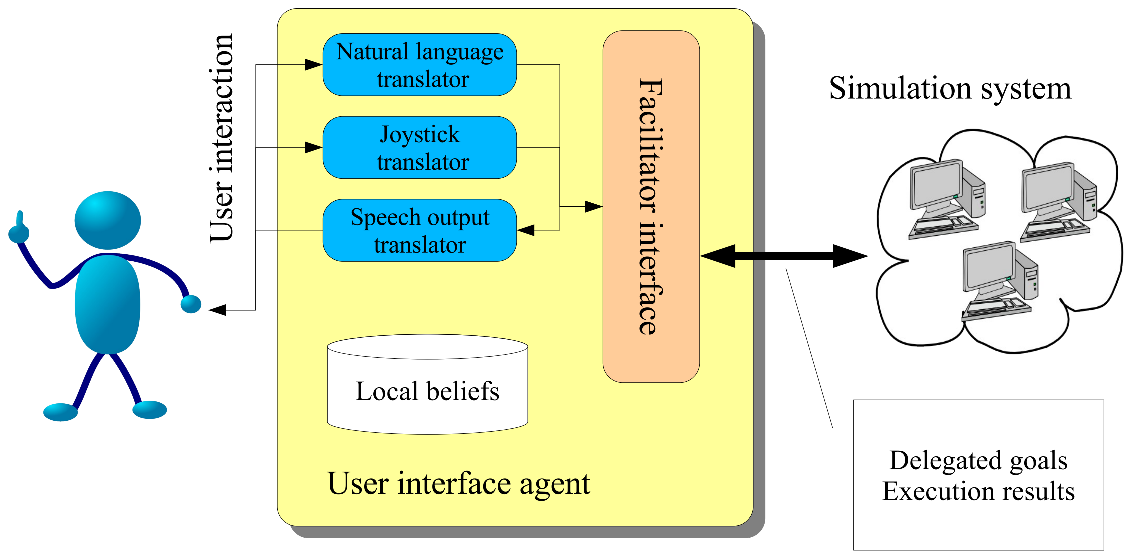


Figure 4.9: The user's involvement in the collaboration environment

From the team building point of view, the Contract Net-based mechanism from section 4.3.2 can also stay the same for the human users building teams to solve problems. After the team is formed the human can either delegate imperative goals to the team members or form a team-oriented plan containing the actions to be executed by the other team members.

Of course, the human user needs some human-oriented interface to the rest of the system, mainly the facilitator. These interfaces are provided by various *translators* mapping the actions of the user to queries and tasks descriptions for the facilitator and the other agents. In the opposite direction a similar translation may be necessary if the user is supposed to receive collaboration requests or results of the tasks he delegated to the other agents. Typically, such human–simulation system interface will be implemented as a specialized agent, providing translation services for the human user on one side and providing a standard agent-like interface for the rest of the system to interact with. A diagram of such an agent is in figure 4.9.

The translators could be compared to the interface agents in RETSINA [99, 35], they have both the same goal – to provide translation of the human input to the machine understandable form. However, there is the difference that in RETSINA the interface agents are using KQML to talk to other agents in the system. Translators typically communicate directly only with the facilitator and the requests of the user are translated into goal specifications to be delegated to the agents.

The human user can participate in the team in two distinct roles:

- As a team leader, the user is responsible for establishing a high level plan how to solve a given problem and for coordinating the team members by delegating them sub-tasks necessary for achieving the solution.

- As a regular team member, the user is responsible for solving whichever task which is delegated to him by a team leader (either another human user or even an autonomous agent). The complex part here is that in order for the delegation protocol to work as designed, the agent representing ("connecting") the human user to the system has to properly declare the user's capabilities. In the case that no capabilities are declared (advertised), the user will be never delegated any task. The solution is to use the translators which have pre-defined capabilities associated with them, such as a gamepad-driven translator declares the capability to navigate the virtual environment.

The last interaction level differs from the normal team work mode in the way the user interacts with the agents in the system. In the regular team work mode, the human user is a part of a collaborating team. In the high level mode, the user interacts with the *whole* teams through their team leaders. The desired goal is achieved by delegating tasks to the team leader who either completes it or proceeds to use the capabilities of the team members to solve it. The activity of the user essentially corresponds to "I want X done, arrange it!" Such way of solving a problem is useful for example for training, where the trainee wants to skip parts he/she is proficient in already and focus only on the new task.

Another example is given in the section 7.5, where a system for declarative "story" creation is described. The "story" is created by the user by specifying events that should happen during the course of the scenario – for example somebody ordering a coffee. It is not important to the user *who* orders the coffee or what is necessary to achieve that, only that the coffee has to be ordered. The agent managing the scenario delegates the request to one of the agents in the virtual environment. The agent solves the given task, managing the issues of summoning a waiter (forming a team), ordering the coffee and having it delivered (tasking the team member) autonomously. The net effect for the user is that the desired goal was achieved, without requiring the user to manage the progress of the solution himself. Typically, user working in the high level mode will have a global overview of the current situation (such as a commander coordinating several smaller units).

To make the role of the interfacing agent simpler, it is possible to allow the full observability of the system for this agent, putting the user into elevated position in the hierarchy of agents. Such extra privileges allow the user access to various information the other agents do not have – such as the declared state, position or other information published by the agents to the facilitator in the form of data and procedural solvables.

By default, the agents have only partial observability of the environment. They have only information which is either internal (produced by them – e.g. their state) or information they acquire by some other means (e.g. sensing, querying the facilitator). Opening up the access to all available information (full observability) for the user can be necessary in certain cases – for example to facilitate creation of a translator which converts the information into graphical map of the current situation or a graphical interface providing the user with detailed information on the agents and the state of the world. A translator with only partial observability would provide incomplete and potentially false information in this case. Another problem could occur if the human user is able to see a situation in the virtual environment which the translator agent is not aware of – the human and the agent do not possess the same senses. Delivery of contradictory information (e.g. the output of the translator vs. what the user sees) is confusing to the user and should be avoided.

4.5 Summary

This chapter presented the proposed collaboration model between autonomous agents and between the autonomous agents and the human users based on task delegation. The model is derived from the Open Agent Architecture with the extensions for supporting explicit teams and different control levels.

A new design for the facilitator agent was shown together with the comparison to the standard OAA facilitator. The two roles (global state keeping and service matching) for the facilitator were analyzed and their implementation described.

Chapter 5

Collaborative problem solving

The ideas presented in the previous chapters allow us to express the state of the world in the symbolic form and to perform basic collaboration based on task delegation. However, for the task delegation to be truly useful for solving practical problems the goal has to be subdivided into smaller parts that the agents can deal with. It is usually achieved by *planning*.

Planning can be defined as a problem solving task, which tries to find a totally or partially ordered sequence of transformations which transform the initial state of the system into the desired goal state. This definition outlines the two types of planners – total order and partial order, which both have application in a collaborative system.

This chapter will focus on the role of the planning in the problem solving process and the problems encountered there. Afterwards, the attention will be given to the problems specific for team planning, where the delegation process has to be considered. The final part of the chapter will be dedicated to the problems of object-specific planning, especially in connection with smart objects.

In direct relation to planning are various contingencies that may occur, such as planning failures (plan not found, planner ran out of memory, etc.) or the possibility of the environment changing while the planning is performed. The possible sources of such contingencies will be noted and analyzed.

One possible approach how to deal with contingencies would be to use *continuous planning* where the agent analyzes and updates the plan continuously during its lifetime, possibly adding and removing goals and updating the perceived state of the world. An example of such continuous iterative planner is ItPlanS described in [34, 63]. The disadvantage of such approach is that the plan is never fully known before executing it (essentially the plan is changing while being executed), making it difficult to determine in advance e.g. the required size of the team needed to accomplish a certain task. This problem led to the decision to work with a standard propositional planner instead and handle the contingencies using fail-safe action execution and re-planning. The implementation details of the contingency handling will be described in the section 6.5.

5.1 Role of the planning in problem solving process

The fundamental goal of a problem solving system is to find a solution to the given problem – usually in the form of a *plan*. The plan is a sequence of actions which have to be performed in certain order to achieve the desired final state of the system (the specified goal). The plan can be also understood as a path in a graph consisting of all possible states of the world (simulation system) from the initial state to the desired goal state. This state-space graph search for the path from the initial state to the goal state is the basis for most of the planners currently in use.

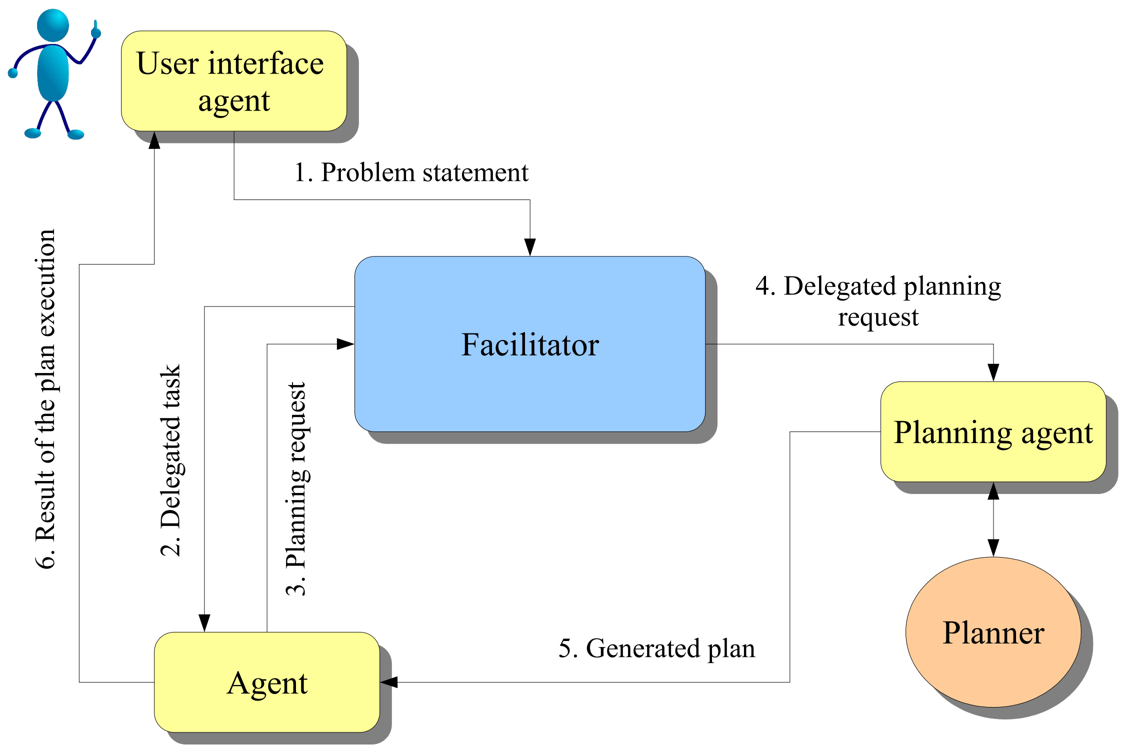


Figure 5.1: Diagram of the planning process

The figure 5.1 shows how the planner is integrated into the problem solving process. First, the problem to be solved has to be specified. Typically, this is done either by the user via his user interface agent or by another agent which needs something solved. The problem is specified in the form of a declarative task/goal specification, as described in the section 3.1.3. The ontology used for defining the goals is application-specific, even though common actions and terms can be reused (such as “move”, “pick-up”, “push”, “agent” etc.). The vocabulary contains terms defined during the development of the particular scenario.

The formulated task specification is delegated to the facilitator for resolution. The facilitator in turn delegates this task further to some agent for solution. The agent creates a planning request out of the current state of the system as it perceives it (using its local beliefs), its capabilities in the form of STRIPS-like operators and the desired goal state. Once the planning request is prepared, it is delegated to the facilitator for dispatching to the planning agents.

Typically, only one planning agent is used, however it is possible to employ multiple planning agents to either have several different planners available for different types of tasks or as a form of load balancing in systems with many agents and large quantity of the planning requests.

Once the planning agent receives the full task description as created by the delegating agent, it creates a problem description for the planner itself. The exact format of the problem description varies with the planner but most of the STRIPS-like planners use some subset of the PDDL language for specifying it. The job of the planning agent is to play an intermediary between a 3rd-party planner and the facilitator-based agent system.

After building the problem description and calling the planner to compute the plan, the results are sent back to the agent originating the request (in accordance to the simplified delegation scheme introduced in the section 4.2.2. The agent then executes the plan one step at a time, both executing the actions in the virtual environment and applying the prescribed effects of the executed actions to the state of the world – both local (agent's beliefs) and global represented by the solvables declared in the facilitator. The actions are executed as a three step process – the preconditions are checked (the environment may have changed since the plan was built), the action is executed and only if the execution was successful the beliefs/world state is updated. In case that the preconditions are not satisfied, failure is reported and the agent then deals with it – for example by replanning or by reporting the failure further.

The final step of the problem solving consists of reporting the results (outcome of the plan execution) to the originator of the task – in the described example it is human user but it can be an autonomous agent as well.

5.2 Problems of the planning approach

Usage of the propositional (STRIPS-like) planning in the teamwork environment with task delegation brings on an unique set of challenges and problems which are typically not present nor addressed in the planning community. Examples of such problems are:

- How to handle task delegation? When is it beneficial to ask another agent for collaboration and when is it simpler/cheaper to do it alone (assuming that it is possible).
- How to deal with the limited knowledge? For example the agent at the planning time may not know what are the identities of the collaborating agents. Thus, it cannot plan for them. However, the plan is supposed to carve out the sub-tasks for the collaborating agents, if needed. Therefore the identities of the agents need to be known. This is a conflicting requirement.
- How to deal with the effects of the delegated tasks on the world? The planner does not know how the other agent is going to perform the task, thus it can assume only very little about the effects of its work. This leads to interesting problems with side effects of actions and inconsistencies.
- Is it possible to use object-specific information in planning? The advantages of such approach are limits on the amount of knowledge needed by the agent itself to be able to solve the problem (some relevant information is provided by the objects themselves) and focus on information relevant to the task at hand – for example agent going to manipulate a box will most likely need information about the box, the information about water being wet is probably irrelevant in such case.

The points outlined there will be now addressed in detail in the following sections.

5.3 Planning with delegation and teamwork

Task delegation and team work create special challenges for the planning process. Typically, the planners are used only for action planning for single agent, not for collaborating teams. In order to be able to use regular STRIPS-like planners in the collaborative environment, special formalism for the collaborative/team actions has to be established.

Another issue that has to be addressed is the problem of forming/disbanding teams. The planner has to be able to form or disband team as needed during the planning process because it is not sufficient to assume that the team is pre-formed before the planning starts. Such teams may very well be completely unusable for the particular task because they may have too few members or members with essential capabilities missing. In the most general case (assuming that there are no readily available "recipes" for specific tasks providing this information) the agent forming the team has no means to know before the plan is created how many agents and with what capabilities it will need for performing it. This information becomes available only during the planning process when the planner arrives to a situation where an action requires an agent to perform it but there is no suitable (satisfying preconditions) agent available according to the state at that planning stage. In such situation new team member needs to be recruited.

A better approach is to let the planner to decide how and when to form the team because the team formation has to be driven by the task requirements. This is a big difference from the systems which use pre-defined teams – in such cases the agent has to work with what it has in the team and its flexibility in finding the solution is limited.

5.3.1 Delegated actions and speculative planning

The formalism of the delegation process is based on the concept of *delegated actions*. From the point of view of the planner, the delegated action is an atomic operator, same as a regular action described in the chapter 3. However, there is a large difference from the agent's point of view. Regular actions (instantiated operators) in the plan are supposed to be executed by the agent itself. The delegated actions are always intended for somebody else to execute, having several consequences both on the planning stage and on the execution stage:

- **The preconditions** of the operator corresponding to the delegated action are usually weaker because the agent doing the planning has very limited or no knowledge about the agent who will the action be delegated to. Therefore, the preconditions cannot be as restrictive as for the regular actions.

  Furthermore, the preconditions explicitly rule out the agent doing the planning as a possible executor of the action. The goal is to delegate the action for execution by another agent if and only if it is not possible for the delegating agent to perform it. From the practical point of view it is not useful to have agents which are not doing anything only delegating the assigned work further. Therefore delegating to self should not happen – the planner should use normal (non-delegated) action instead and this is achieved by this restriction.

  Finally, the last condition for the delegated action to be possible is that the delegating agent has to be a team leader. The reason for this seemingly artificial restriction is also clear – if the restriction was not in place, the agent A could delegate the task to agent B which in turn can delegate the same task to agent C etc. Such behavior, while valid, is obviously not desirable as the user is rarely interested in the agents "playing ping-pong" with the task.

- **The effects** of the operator are limited. Only the effects which are desired and known are present. However, performing the action at the execution time can have other, unknown in advance, side effects which cannot be accounted for at the planning time.[1] There are several reasons for this, the most common one is that the agent which received the delegated action for execution has a different perception of the virtual world than the delegating agent (since they do not share their beliefs). There are various consequences of this fact – for example the delegated action may have a side effect (additional action executed) which is required in order to account for the change in the environment the team leader was not aware of. This side effect may cause a failure of the remaining actions in the plan at a later stage, because it may have done other changes to the environment which were not accounted for in the original plan.

Alternatively, the receiving agent could have used the services of the planner too to determine how to perform it. That means that several actions will be executed instead of just one and the delegating agent has no means to know this at the planning time.

Both of these issues mean that the planning with delegated actions is a best-effort case. The information the planner has at disposal at the moment when the planning is performed can never be complete enough to warrant that the plan generated will not fail during the execution stage. Such planning is *speculative* and *optimistic* – the planner attempts to work with incomplete data (therefore speculative) and assumes that the actions declared as possible are truly possible (that may not be the case at execution time, therefore optimistic).

Figure 5.2 shows two planning operators – a regular `move` action and its `delegated-move` delegated counterpart. The differences outlined in the previous paragraphs are clearly visible – the delegated action does not have the restriction that the two places between which the movement is done have to be connected (by a walkable space, for example). The rationale is that the delegating agent does not care about such detail, it is the responsibility of the receiving agent to figure out how to move between the two places. It may require, for example, that a door has to be opened or an obstacle removed.

Such additional action will change the world state and produce side effect which the delegated action cannot account for. If it was a regular `move` action, such side effects could not occur – they would have to be effects of some previous regular action in the plan because otherwise the regular action would not be applicable. For example, because the door is opened by some action, the two places become `connected` and it is possible to generate the `move` action in the plan because the corresponding precondition is satisfied.

The special symbol `self` is an instance of a *protoagent*, symbolically denoting the agent performing the planning. The concept of protoagents will be described in the next section.

[1]The HTN-based planners distinguish also between *primary* and *secondary* effects. Primary effects are used to achieve the goals, secondary ones are used only during precondition testing but cannot be used to achieve goal. The difference with side effects is that the secondary effects are *known* at planning time, however the side effects will be known only at execution time of the plan – the planner has no information about them and cannot use them during the planning process.

```
(:action move
 :parameters (?who ?from ?to)
 :precondition  (and (at ?who ?from)
                     (not (= ?from ?to))
                     (place ?from) (place ?to)
                     (agent ?who)
                     (not (busy ?who))
                     (connected ?from ?to)
                     (= ?who self))

 :effect  (and (at ?who ?to)
               (not (at ?who ?from)))
)

(:action delegated-move
 :parameters (?who ?from ?to)
 :precondition  (and (at ?who ?from)
                     (not (= ?from ?to))
                     (place ?from)
                     (place ?to)
                     (agent ?who)
                     (not (busy ?who))
                     (not (= ?who self))
                     (teamleader self))

 :effect  (and (at ?who ?to)
               (not (at ?who ?from)))
)
```

Figure 5.2: Regular versus delegated version of the *move* action

5.3.2 Generic plans

In order to be able to use the delegated actions to offload work to the other agents, the planner needs to know the identities and capabilities of the agents available for collaboration. In the facilitator-based system described in chapter 4 this information may not be generally available to the agent. The idea of "delegated computing" is to exactly avoid the need for the agent to know this kind of information in order to simplify the agent and to improve its domain independence.

In such a system the required information can be gained in two ways:

- The team is pre-formed.
- The agent learns the identities of the team members at the execution time, after forming the team using the Contract Net protocol.

The first case is not desirable, as described before (the teams should be dynamically created in order to be able to adapt to different tasks). The second case does not work until the plan is created and executed which is too late for the planner. To address this catch-22 situation, new concept of a *protoagent* is proposed.

Protoagents are placeholder agents. The closest analogy to a protoagent is a formal parameter to a function call – the parameter is used inside the function, however its real value is determined at run-time, when the function is called with the real values substituted for the formal parameters. Protoagents are used in a similar way – the planner plans using protoagents in place of the real identities of the agents involved. The plan which gets created by the planning agent is therefore *generic*, independent from the agents available. At the execution time, the protoagents are replaced by the identities of the agents which are forming the team – the generic plan is instantiated and executed.

The protoagent together with the required capabilities (the actions the protoagent is asked to perform in the given plan) describe the *role* of the agent in a team. The purpose of the protoagent itself is only to define the identity of the team member for the planner and later to map it to the identity of a real agent.

The generic nature of the produced plans allows also plan reuse – the same plan solving certain goal can be reused in a different situation by a different team of agents without change (assuming that the current state of the world matches the initial state required by the plan). Such plan reuse can significantly save planning time for frequently occurring tasks, because the plans can be cached and reused as needed instead of generating them each time anew.

There are several rules governing the use of the protoagents. These rules define the collaboration rules that the planner can utilize in order to solve the given goal. All these rules are implemented in the form of special STRIPS operators and preconditions/effects added to other operators.

1. Protoagents are named $AI, I \in [0, 1, ...n)$, n is usually some small positive integer defining the upper limit of the possible team size. In PDDL syntax the protoagents are written as `A1, A2, A3, ....`
2. Protoagent `self` denotes the agent which requested the computation of the plan.
3. The predicate `(agent self)` evaluates always to true (i.e. this agent is always "recruited" and available).
4. A regular action can be instantiated only for the protoagent `self`.
5. The fact `(agent ?a)` is defined for the protoagent `A` by the operator `(recruit-help ?who ?protoagent)`, where `?who` is the team leader (usually `self`).
6. The fact `(agent ?a)` is negated/undefined for the protoagent `A` by the operator `(disband-team ?who ?protoagent)`, where `?who` is the team leader (usually `self`).
7. Delegated action operators can be instantiated only for a protoagent for which the predicate `(agent ?a)` evaluates to true in the given state of the planner.

The interesting cases are the last three rules. Rule 5 says, that a protoagent can be "changed" into an agent by using a `(recruit-help ...)` operator and conversely the rule 6 says that this change can be reversed by a complementary operator `(disband-team ...)`. These two operators implement the Contract Net team forming protocol from the point of view of the planner.

The complex three-way handshake from the section 4.3.1 is hidden behind the `(recruit-help ...)` operator. For the planner this is an atomic operation which produces one agent ready to use in the planning process. The protoagents are the "material" for this process. At the execution time, the agent executing such plan will try to form team or to add new team member to its existing team.

Rule 6 indicates that a delegated action operator can be instantiated only for such protoagent, which was made into a full agent. That is possible if and only if the protoagent was "recruited" before. The rationale is that the sub-tasks should be delegated only to valid team members, not to arbitrary agents. From the implementation point of view, this constraint ensures that the agent's identity will be known at run time in order to delegate the sub-task to it.

5.3.3 Planning with delegated operations

From the planning point of view, the delegated operations introduce several interesting problems. As explained in the section 5.3.1 already, the planning process becomes both speculative and optimistic because of the ultimately incomplete information that is available to the planner.

Apart of having to deal with the consequences of having an incomplete information, the issue of deciding when to delegate a task to the collaborating team members and when to tackle the problem alone still remains. This task is of great importance if a reasonably working team is to be achieved. In our case, this decision has to be made at the planning time, when the solution for the solved task is being found.

The problem can be reduced to a graph search problem, where we are looking for the best path between the start and the end node. The notion of being the "best" is usually defined using a cost function $f(n)$ for the graph node n. A typical example of this approach is the A^* algorithm frequently used for path planning (finding of the shortest path between two points). In A^* the cost function has the form:

$$f(n) = g(n) + h(n)$$

Where $g(n)$ is the cost of the path in the graph from the start node until the node n and $h(n)$ is an *estimated* cost of the path to the end (goal) node. The estimate is computed using a heuristics – for example in the 2D path planning case the Euclidean distance between the node n and the goal node is frequently used.

In this context the propositional planning process can be described as a search in the space of all possible actions, where the nodes of the graph are instances of the planning operators (actions) and the edges of the graph are linking together possible actions. This is a very simplified description, for the full definition of the planning graph approach to planning see [10].

This simplified description still suffices for the purpose of introducing the total cost of the plan as its *length* – amount of steps needed to transform the initial state into the goal state. The steps are considered to have unit cost.[2] Most propositional planners are trying to find a shortest plan possible for the given initial state and the given goal.

By exploiting this property, it is possible to tell the planner which actions are preferred – such as that it should prefer always the non-delegated actions compared to the delegated ones. To practically achieve this, one has to ensure that the cost of the plan generated for the preferred action is shorter

[2] Of course, all action do not have the same cost (resource usage, duration, etc.) and this approach can lead to unrealistic plans (e.g. a single but very expensive action gets scheduled instead of multiple but cheaper ones). Planning with resources/time can be simulated by using numeric fluents, examples of this approach are well known in literature (e.g. the numeric variants of the Depots, Satellite domains available at `http://planning.cis.strath.ac.uk/competition/`).

than for the other actions leading to the same goal. In most of the simple cases, the delegated action is more expensive in terms of the plan length because of the team creation overhead. There has to be at least one extra action for recruiting the team member compared to the non-delegated action. This automatically ensures that the non-delegated actions will be preferred by the planner if there is a choice.

However, one has to consider the possibility of when the cost of the delegated action becomes lower than a cost of equivalent sequence of non-delegated actions. Let's assume a simple virtual world having a topology as shown in figure 5.3.

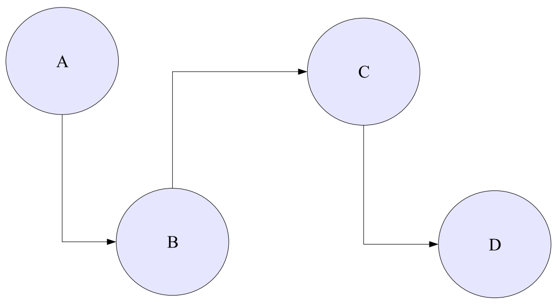


Figure 5.3: Simple world topology

In this world, the cost to get from place *A* to place *D* is 3 because three steps are needed. The plan looks like this:

```
(move self A B)
(move self B C)
(move self C D)
```

However, let's assume, that it is possible for the agent to summon a car with a driver (recruit it into a team) and then tell it to drive him to the destination (delegated action). The resulting plan may look like this:

```
(recruit-help self a1)
(delegated-drive a1 self A D)
```

This plan is one step shorter than the original plan which used only non-delegated actions. That means that this plan will be generated by the planner instead of the original three-step one, because the search will be able to finish sooner (the plan is shorter, the A^* heuristics guides the search to the shortest branches first). A real world meaning of this is that if the cost of moving between places in the virtual world is more than two steps (a break even case), the agent will prefer to be driven around instead of walking if a car is available. This is actually desired behavior in this case.

Of course, the cost of performing the `delegated-drive` action by the agent `a1` may be higher than original three step plan using only the non-delegated actions. However this cost is *hidden* from the planner because the team leader does not care how this action is performed. That also means that care has to be exercised when designing the delegated actions in order to avoid absurd situations – such as the execution of the delegated action costing much more than the non-delegated option (e.g.

in terms of execution time, used up fuel, etc.). This is obviously a domain-dependent issue that cannot be addressed in generic way. By default, the delegated actions cost at least one unit (step) more than non-delegated ones because of the agent recruiting step necessary to create the team before being able to delegate the task.

Original `delegate-drive` action, cost 1.

```
(:action delegated-drive
 :parameters (?who ?passenger ?from ?to)
 :precondition  (...)

 :effect  (and (at ?who ?to)
               (not (at ?who ?from)))
)
```

Collection of `delegated-drive-step-?` actions, cost 2

```
(:action delegated-drive-step1
 :parameters (?who ?passenger ?from ?to)
 :precondition  (...)

 :effect  (delegated-drive-step1 ?who ?passenger ?from ?to)
)

(:action delegated-drive-step2
 :parameters (?who ?passenger ?from ?to)
 :precondition
      (and (delegated-drive-step1 ?who ?passenger ?from ?to)
            ...
           )

 :effect
      (and (at ?who ?to)
           (not (at ?who ?from))
           (not (delegated-drive-step1 ?who ?passenger ?from ?to)))
)
```

Figure 5.4: Action splitting to explicitly increase cost

If necessary, the cost of an action can be artificially increased by inserting synthetic[3] actions – the problematic action can be split into two, where the first one retains the preconditions of the original action and has a synthetic effect creating an intermediate state. Then a second action is introduced, which requires this intermediate state in its preconditions and has the effects of the original action. This process can be repeated as needed, in effect assigning an explicit cost to the action (with the default cost of an action being 1), even though the planner itself does not support action costs. This

[3]Synthetic in the sense that these actions do not serve any other purpose except of increasing the length of the plan.

process is illustrated in the figure 5.4, where a simplified action `delegate-drive` is split in order to increase its cost by 1.

The advantage of using action splitting over explicitly introducing costs as e.g. numeric fluents is in the wider choice of planners. Not every planner supports numeric fluents (e.g. original Graphplan or pure STRIPS-only planners in general), but the action-splitting approach will still work without modification. This technique is valuable especially in cases when the planner available has other useful features but is missing support for numeric fluents – such as being partial-order planner or ability to generate contingent plans (e.g. Sensory Graphplan).

5.3.4 Multi-stage planning in teams

Delegated actions allow distribution of the workload among the collaborating agents formed in the team. The action planning process from the team leader's point of view was described in the previous section already. From the team leader's point of view (and from the planner's point of view as well) the delegated action is an opaque black box promising to change the world state as declared by its effects. This information, together with the delegated actions' preconditions, is sufficient for the planner to create the high level plan for the team leader. However, this information does not specify at all how the team members are supposed to perform the assigned tasks. This responsibility is delegated to them together with the assigned task.

In order to solve the delegated task the delegate has to determine how to do so. In fact, the task being delegated is specified in exactly the same form as the task the team leader received – in a declarative form specifying the desired state of the world. It is only natural to use the same process to solve the delegated problem – namely to use the planner in the same way as the team leader did. This leads to new planning stage and the process may repeat in case that the delegate needs to delegate another sub-task of the given goal further.

For practical reasons, the applications described in the section 6 consider only three level hierarchy maximum – the task originator (may be human user or another agent), the team leader and the team members. The tasks can be delegated from the originator to the team leader and then sub-tasks from the team leader to the team members. The limit is arbitrary, there is no theoretical reason why the team members couldn't become team leaders of their own teams solving the assigned sub-task, but it would create lot of complications in the implementation with little practical benefit in the common VR domains (collaboration with single or multiple virtual humans, delegation of a task to the team of virtual humans).

In order to be able to generalize the structure to more than three hierarchical levels, changes would have to be made in the task delegation and team forming process. For example, the agent could play multiple roles – team member receiving tasks from the team leader, team leader of its own sub-team and team member of a sub-team formed by another agent. It would be necessary to ensure that there are no "conflicts of interest" between the activities of these teams (i.e. no conflicting goals) and a conflict resolution process would need to be introduced. Furthermore, it would be necessary to have an ability to abandon commitments. In the presented architecture, the commitment is very simple – the agent commits to the team activity regardless of what it will be (the agent commits blindly). However, in order to resolve conflicts, the agent would need either to know in advance what tasks are to be performed so that it could decide whether or not to commit to performing them (e.g. by detecting a conflict with another set of tasks from another team leader) or to have an ability to abandon commitment if a conflict is detected. This could in turn cause plan failures (the plans are assuming that the task will be performed, not abandoned) and the agents would have to be able to deal with this additional workload. This extension is considered for the future, but was not implemented for this thesis.

Planning schemes with similar hierarchical work distribution are known in literature as *hierarchical planning* and are usually used to limit the amount of work the planner has to do at each level of the hierarchy. However in most cases the hierarchy is fixed a priori – for example the chain of command in military simulations. In the described system, the hierarchy is dynamic and created as needed, totally depending on the current state of the world and the task being solved.

The multi-stage planning performed by the delegates can bring problems to the team leader, unfortunately. Usually, there is little or no coordination of activities between the team members solving different tasks except by the team leader executing the high level plan. Because the actions of the team members may have side effects which the team leader cannot predict, conflicts may occur. These have to be resolved at run time, usually by sensing the changed environment and re-planning. In most cases it will be sufficient to re-plan the failed sub-task only, in the worst case the whole high-level plan has to be re-planned, potentially failing again.

In order to avoid the expensive re-planning as much as possible, it is possible to pass on information from the team leader to the team members which can help them to solve their assigned tasks. However, there is no general way how to determine which information will be useful for the agent in advance, before he creates the plan, thus this remains to be addressed in case-by-case application specific manner. There is no way how to completely prevent the task failures and the ensuing re-planning, however.

5.4 Object-specific planning

In the previous sections, the agents were using a planning system combined with delegated actions to solve given tasks. For such a setup to work properly there are two main requirements:

1. The agents know the possible actions (operators).
2. The agents have enough information about the current world state to know which operators are applicable.

These two requirements are really basic but they can put a large strain on the planning system. It is possible to have many actions available (the prototype has more than 30) and many predicates describing the current state (the bar scenario in the section 7.5 contains around 500 predicates in the beliefs of each agent and this is still a small scenario). Such large amount of information that has to be processed by the planner leads to long planning times and undesirable delays in the animation while the system is "thinking".

To mitigate this problem, there are two possible solutions – either a faster planner or reduction of the information to only the data relevant to the task. Ideally, both paths should be pursued.

Reduction of the irrelevant predicates from the initial state is possible and was described in literature (e.g. in [22, 75]). The goal is to discover which initial facts are relevant to the solved problem – in other words which allow the operators contributing to the goal to be applied – and remove the rest from the initial state. This approach often allows for dramatic speed-ups, especially for planners like Graphplan where the size of the data structures grows exponentially with the size of the initial state.

Another option how to reduce the complexity of the planning is to not have the irrelevant operators (actions). If it would be possible to let the agent learn the actions specific to objects on the fly at run time, it would need only few generic actions, like moving or picking up objects and the remaining specialized actions could be "learned" only when needed. This is a main idea behind object-specific planning – the plan is created using object-specific actions which are not used for anything else and retrieved at run time.

In order to enable the object-specific planning, the information about object properties and object-specific actions has to be somehow tied to the object. This issue was solved by extending an existing *smart object* framework to encode also the *semantic information* needed by the planner together with the corresponding animation code. The description of the basic smart object concept can be found in [51, 4].

Smart objects provide not only the geometric information necessary for displaying them on the screen, but also semantic information needed for animation purposes. This information is stored in the form of sets of attributes attached to the geometry of the object – such as important places on or around the object (e.g. where and how to position the hands of the virtual character in order to grasp it), animation sequences (e.g. a door opening) and general, non-geometric information associated with the object (e.g. weight or material properties).

Smart objects also contain "interaction plans", which are essentially scripts containing the animation of the action itself. These scripts coordinate the human and object animations to create the intended result, which could be a virtual human pushing a crate, opening a door, etc.

In order for the smart objects to be usable for planning, it is necessary to extend the animation-oriented data with semantic information conveying the knowledge *what can be done* with the smart object, what are the *requirements* and what are the *consequences* of such actions. This matches very well the definition of the planning operators and the extension is therefore straightforward – for each object-specific action that can be performed on the smart object, the following information is added using a set of predefined attributes:

- Name of the operator
- Operator definition
- Animation script for this operator

On top of this information a set of predicates and numeric fluents expressing properties specific to the object may be added. This additional information is added to the agent's beliefs when the object is queried.

The operator definition define the operator in terms of its arguments, preconditions and effects, as described in the chapter 3. By examining the attributes of the smart object, the agent is able to discover the object-specific operators and properties and add them to its beliefs. The semantic information embedded in the object plays a role of an "operation manual", defining to the agent how the object works and what is it capable of.

The planning process using the object specific information works as depicted in the figure 5.5:

1. The agent collects the information from the smart object and adds it to its beliefs.
2. The problem description in the PDDL form is built by the agent, as with regular planning.
3. The planning is performed, now using the specific information retrieved from the smart object as well.
4. The plan is executed. In case that an object-specific action is encountered in the plan, the corresponding animation script is retrieved from the object and executed.

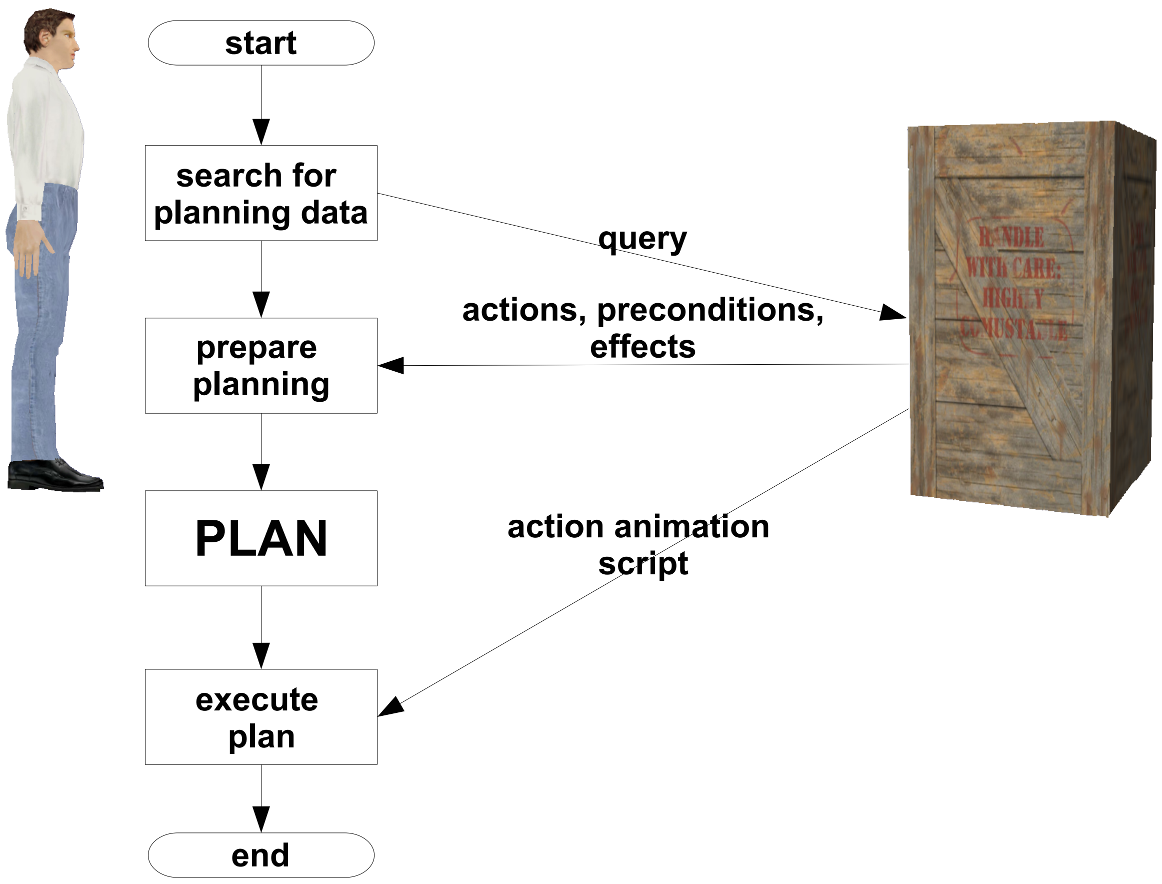


Figure 5.5: Planning with object-specific actions

To illustrate the process, let us consider an example of a virtual character operating a jukebox in a bar. This scenario is present in the case study described in section 7.5 and the associated animation issues are described in the section 6.1.2. The virtual character wants to dance in the bar. One of the preconditions of the `dance` operator (fig. 5.6) is that jukebox has to be powered on. According to the algorithm outlined in the figure 5.5, the agent will first collect the information from the smart object, in this case jukebox. The jukebox object contains the object-specific action shown in figure 5.7.

The problem description is built, using the operator retrieved from the jukebox smart object. Standard planning is performed and new plan is generated which will use the object-specific action from the smart object to achieve the goal. Upon execution, the object-specific action is performed using the generic action mechanism, described in the section 6.1.2.

Object-specific actions have several advantages. They reduce the amount of operators the agents have to know about and reduce the amount of irrelevant information about objects in general. The object-specific information is retrieved only when needed. Another advantage is that the smart objects are completely self-contained. They contain all information that is necessary for the agents to use them, both from the high level planning point of view and from the low level animation side as well (see [4, 51]). This property enables better reusability of data because it is sufficient to load a new smart object into the virtual environment and the agents will automatically "learn" how to use it.

Compared to other works, such as Perlin's Improv, Badler's Jack [34], work of Vosinakis [110] or work of Levison [62], the smart object approach moves part of the processing and information to

```
(:action dance
 :parameters (?who ?jukebox ?where)
 :precondition (and
                     (agent ?who)
                     (place ?where)
                     (dance-place ?where)
                     (jukebox ?jukebox)
                     (at ?who ?where)
                     (powered-on ?jukebox))

 :effect  (increase (THIRST) 20)
)
```

Figure 5.6: The `dance` operator

```
(:action power-up-jukebox
 :parameters (?who ?what)
 :precondition (and
                     (agent ?who)
                     (near ?who ?what)
                     (machine ?what)
                        ; (machine ?m) -> can be turned on/off, more
                        ;                 general than jukebox

                     (not (busy ?who))
                     (not (powered-on ?what)))

 :effect  (powered-on ?what)
)
```

Figure 5.7: The object-specific `power-up-jukebox` operator

the objects being manipulated. The agents do not need to have and maintain information about all the objects in the scene, the information is decentralized. The extension described here adapts this approach to high level reasoning-related information too.

5.5 Summary

The most important contribution presented in this chapter is the concept of delegated actions and their integration into the planning process. The delegated actions allow the agent to reason about the actions of its team members and plan for them.

A concept of generic plans was introduced together with the new notion of protoagents addressing the problem of team planning in OAA-like system where the identities of the collaborating agents are not available a priori. These two concepts provide a powerful tool to the agent, enabling it to reason about actions of a team before the team is actually formed and without any modifications to the planner necessary.

Finally, the idea of object-specific planning was proposed exploiting the distributed semantic information stored in the smart objects. Object-specific planning enables the agents to reason about objects never before encountered and to learn new “skills” at run-time by examining them.

Chapter 6

Multi-agent simulation framework

The previous chapters focused on the theoretical aspects behind the work presented in this thesis document – how to represent a virtual world, how to establish collaboration between the human user and the autonomous agents (or the agents themselves) and finally how a problem can be solved using teams of collaborating agents.

This chapter will present results of this research in the form of a multi-agent simulation framework based on the presented principles. This framework was used to implement six case studies to evaluate the feasibility and usability of the techniques proposed in this document.

6.1 Overall architecture of the implementation

The overall design of the implemented simulation framework is depicted in the figure 6.1. The system consists of several key components which are linked together using CORBA interfaces:

- Visualization and animation engine
- Puppet abstraction layer
- Ghosts
- Facilitator
- Grid agent
- Location agent
- Planning agent(s)
- Application-specific agents

The visualization and animation engine is responsible for displaying and animating the virtual world and its presentation to the user in some form – for example using HMD[1], large projection screen or computer display. The current implementation uses either VRlab's VHD++ framework (described in detail in [82]) or the Delta3D engine[2]. Both engines were equipped with CORBA interfaces to expose their functionality to the external components.

[1]Head Mounted Display.

[2]Available from `http://www.delta3d.org/`.

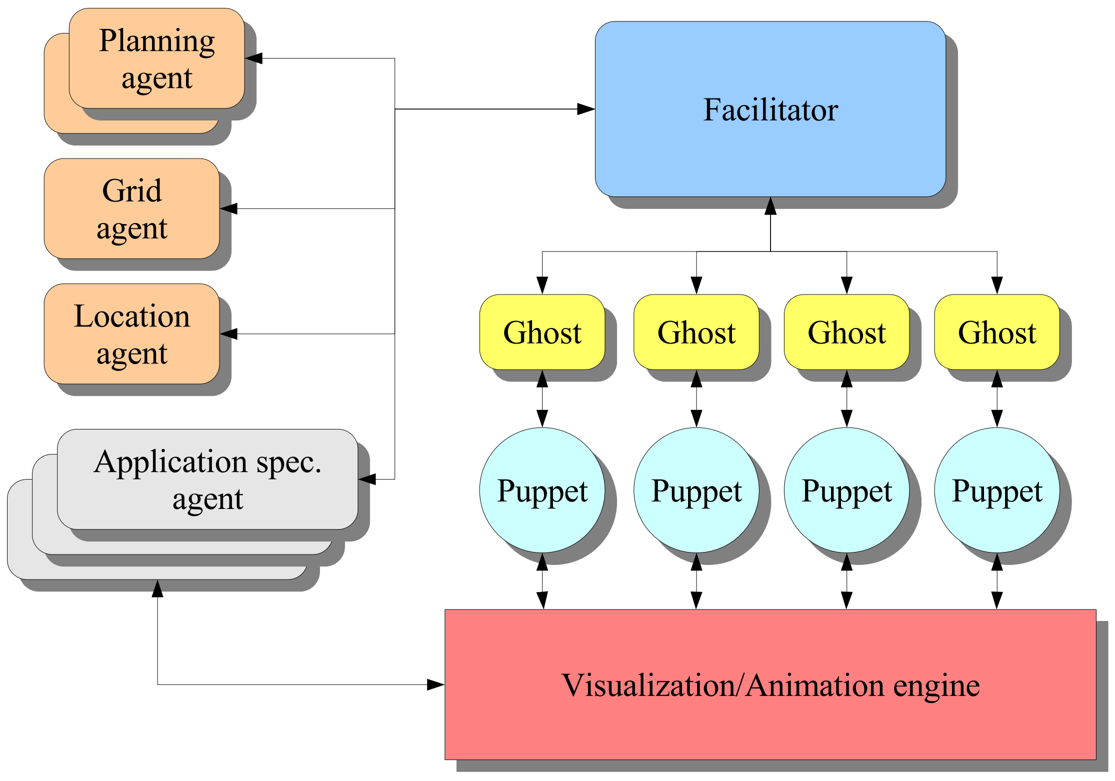


Figure 6.1: Architecture overview

The puppet abstraction layer is responsible for shielding the rest of the framework from the animation engine specific issues, such as how to trigger animation actions, how to move objects in the scene etc. At the same time it integrates the primitive atomic actions offered by the 3D engine into higher level actions – for example an action for opening a door by a virtual human character may consists of several animation primitives (move to the door, grasp the handle, pull the handle, un-grasp the handle) which are combined into a single logical action.

The ghosts are the core of the simulation, they are used to implement the behavior of the individual agents. In the most common case the ghost represents an agent controlling a single puppet which in turns animates a single virtual character in the virtual environment. Such arrangement represents a single virtual character having both the body (geometry in the animation engine), motor control (puppet) and mind (the ghost). Another typical use for a ghost is an agent which implements part of an interface for a human user used to control the user's avatar in the virtual environment.

The puppet abstraction layer and the ghosts form an important subsystem in the described framework. Their design and inner workings will be described in detail in the section 6.1.2.

The facilitator is the central part of the framework, enabling collaboration between the agents and keeping the state of the virtual world. Because of its central role in the system, the implementation of an efficient facilitator is of great importance. The facilitator will be described in detail in the section 6.2.

Grid and location agents provide auxiliary services to other agents, such as resolution of symbolic names of places into coordinates and path planning in the virtual environment. They are implemented in the form of agents and not just services in order for the other agents to be able to use the task delegation to dynamically find them through the facilitator and to delegate the specialized tasks to them (such as path planning, environment queries etc.). They perform similar role as task agents in RETSINA [99].

Planning agents are similar to the grid and location agents in the way they communicate with the rest of the system (high-level communication through delegation via the facilitator). They provide the reasoning capabilities to the other agents participating in the system. Planning is used to build plans solving problems encountered by the agents and also to prescribe how to collaborate with others in the process. The basic concepts were described in the previous chapters, the section 6.4 will focus on the implementation details of the planning subsystem.

The final component are various application-specific agents implementing application-specific services, such as various user interfaces, monitoring agents or provide access to specialized resources. These agents are not really part of the framework because they are application dependent and specific to each setup.

6.1.1 Basic technologies

All mentioned components except the 3D engine were implemented in the prototype version of the simulation framework using Python language. Python[3] is a high level, loosely typed language allowing fast implementation of complex applications compared to lower level compiled languages such as C++ or Java. It could be argued that its semi-interpreted character (the code is compiled into bytecode and interpreted afterward, similar to Java) makes it unsuitable for time-critical applications such as VR simulation platform, however in practice this is a non-issue. The bottlenecks are frequently elsewhere, Python is very rarely a problem – e.g. if a virtual character has to walk a longer distance it will take the same time regardless whether the order to walk was given by Python or C++ code, but the Python code will be much simpler and faster to write.

As mentioned before, the individual components are communicating together using CORBA interfaces. Even though the inter-agent communication is essentially message-based (requests and replies are sent and queued as encoded PDDL expressions using immediately returning CORBA calls), it was implemented on top of CORBA method calls, using standards such as COS Event Service for message passing.

There were multiple reasons for selecting CORBA over a homegrown solution or some other tool, namely:

- Multi-process communication – CORBA objects are typically standalone processes, enabling simpler implementation of the agents and system components and minimizing risks of undesired interactions between components which are frequent in monolithic systems.
- Network transparency – CORBA allows seamless distribution of the objects over the network. This feature was very important at the design stage because it allows to harness the computational power of multiple machines without having to do any change to the application itself. The simulation system can be scaled up or down as needed by adding or removing computers, from a single laptop running everything to a large cluster of networked machines.
- Language independence – CORBA is available with almost every major programming language, making the system easy to extend with heterogeneous components written in different programming languages.

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

- CORBA is an industry standard – even for a research project this fact is valuable because it means that the tool is proven and well supported by development tools.

The implementation uses omniORB[4] implementation of the CORBA standard, together with its excellent Python bindings. This implementation is interoperable with other ORBs, we have successfully tested Java ORB or MICO[5]. The omniORB implementation was chosen mainly because of the availability of supported Python bindings and its simplicity for both C++ and Python.

6.1.2 Ghosts & puppets framework

The ghost and puppets framework is one of the most important components of the simulation system. Its role is to act as a bridge between the low level animation code and high level artificial intelligence software. There are few examples in the literature how such functionality can be implemented because it falls outside of the scope of a typical animation or AI research.

Some attempts at such interface between low level animation and higher level reasoning can be found in the work of Blumberg [11], Perlin [80, 81], Terzopoulos [33, 108] and others. However, except of Blumberg, none of them deals with issues such as controllability or control sharing. Blumberg in his thesis [11] outlines multiple control levels possible for the virtual character, together with the possibility to integrate external control, such as input of the movie director. Compared to his work, the work presented here focuses more on the high-level issues, such as reasoning and team work, not so much on the animation and behavior problems he was addressing.

A new design was created with several goals in mind:

- Controllability – the ability of the user to control the simulation has priority over autonomy of the agents. Controllability and determinism of the simulation is a frequent requirement especially in the training scenarios. Fully autonomous agents work well for artificial life simulations, however they make the scenario difficult to control.

- Control sharing – the ability for multiple agents to control the same object. This ability is useful especially for human avatars where some part of the functionality is controlled by the human user and some parts are controlled by an autonomous agent. For example, a human user may wish to control the camera while his avatar is walking under the control of an agent towards next objective.

- Information access – it has to be possible to interrogate the internal state of the objects in the virtual world. This is typically achieved in a low level, animation engine dependent way and a suitable abstraction is needed.

- 3D engine abstraction – different problems have different solutions and different needs. There is no “silver-bullet” solution for everything, thus the ability to swap the animation engine for another one is an advantage.

  Some engines are suitable for high-detail simulations with few agents (such is the case of VRlab’s VHD++), but for example for crowd simulations containing thousands of agents a specialized solution is required.

[4] `http://omniorb.sourceforge.net/`
[5] `http://www.mico.org/index.html`

Puppets: Puppets are a special type of software agent acting as an intermediary between the high level AI and low level animation system. The name "puppet" comes from an important property – the puppet does not have autonomous behavior (i.e. it is a software agent but not an autonomous agent in the AI sense). It is always controlled by some external entity – a ghost. A ghost can assert control over the puppet – *possess* it. This control assertion analogy gave the name to the framework.

The protocol driving the process of exchange of control over one puppet is very simple. Figure 6.2 documents it. One of the ghosts has the control over certain part of the puppet. Each puppet may consists of several parts which can be controlled by different agents. The reason for such subdivision is division of work – for example a vehicle may require several agents to collaborate to operate it or a large crate may need two agents to move it because of its weight.

Once the other ghost tries to take control (possess) the same part of the puppet, it will get an exception, reporting failure. The ghost may now opt to notify the incumbent one about its desire to take the control. A properly behaving ghost is required to periodically test, whether there is a notification flag set in the puppet and relinquish the control if it is the case.

This design is not really optimal, ideally an asynchronous mechanism such as events or callbacks should be used instead of polling. Polling was implemented because it didn't require any additional external event broker at this stage. In hindsight, proper event-based implementation would have been better, even though the practical performance impact is low – the control is exchanged rarely and the single check per simulation frame is not costly.

Moreover, the mechanism does not protect against rogue ghosts not behaving properly (not freeing the puppet on demand). The ghosts have to be cooperative, the protocol cannot protect against this. The main reason for making the protocol cooperative like this was the issue of consistency – if the ghost gets removed from control of the puppet in the middle of an action, it could leave the puppet and the underlying engine in undefined state (e.g. in the middle of an animation). Complex state saving and restoration would be necessary, both on the side of the puppet and on the side of the 3D engine where this is really difficult to do – e.g. stopping an iterative inverse kinematics process in the middle and then restarting it later without seeing artifacts and taking into account the potentially modified posture of the character is very complex task. Not to mention that the character could have been performing multiple animations at the same time, not only one, making the problem untractable in practice. As a workaround for this problem, the ghosts will release the control of the puppet if and only if the action they were performing is finished, never during the duration of the action. This has practical consequences, e.g. the character will not react to the request for taking over control immediately but only after it finished the current action.

Finally, after the control is released, the other ghost can take over and perform some actions. During this period the original ghost is waiting and periodically trying to take the control back ($\approx$ every 5 seconds). This succeeds only when the ghost in control releases the puppet and then the original ghost can continue with the original task it was performing.

This schema is especially useful in case that a human user wants to intervene because the autonomous agent (ghost) controlling the puppet is misbehaving in some way. The user can use a special ghost to remove the original agent from control, take the puppet over and for example move the virtual character to the desired place or perform some action. After releasing the control the original ghost can take over and continue with the original activity.

The described control swapping scheme satisfies the goals of control sharing and also of controllability – the user is always allowed to intervene (by replacing the ghosts) in case that he/she desires to and has always the option to influence the scenario as needed. This is an important feature for many training scenarios.

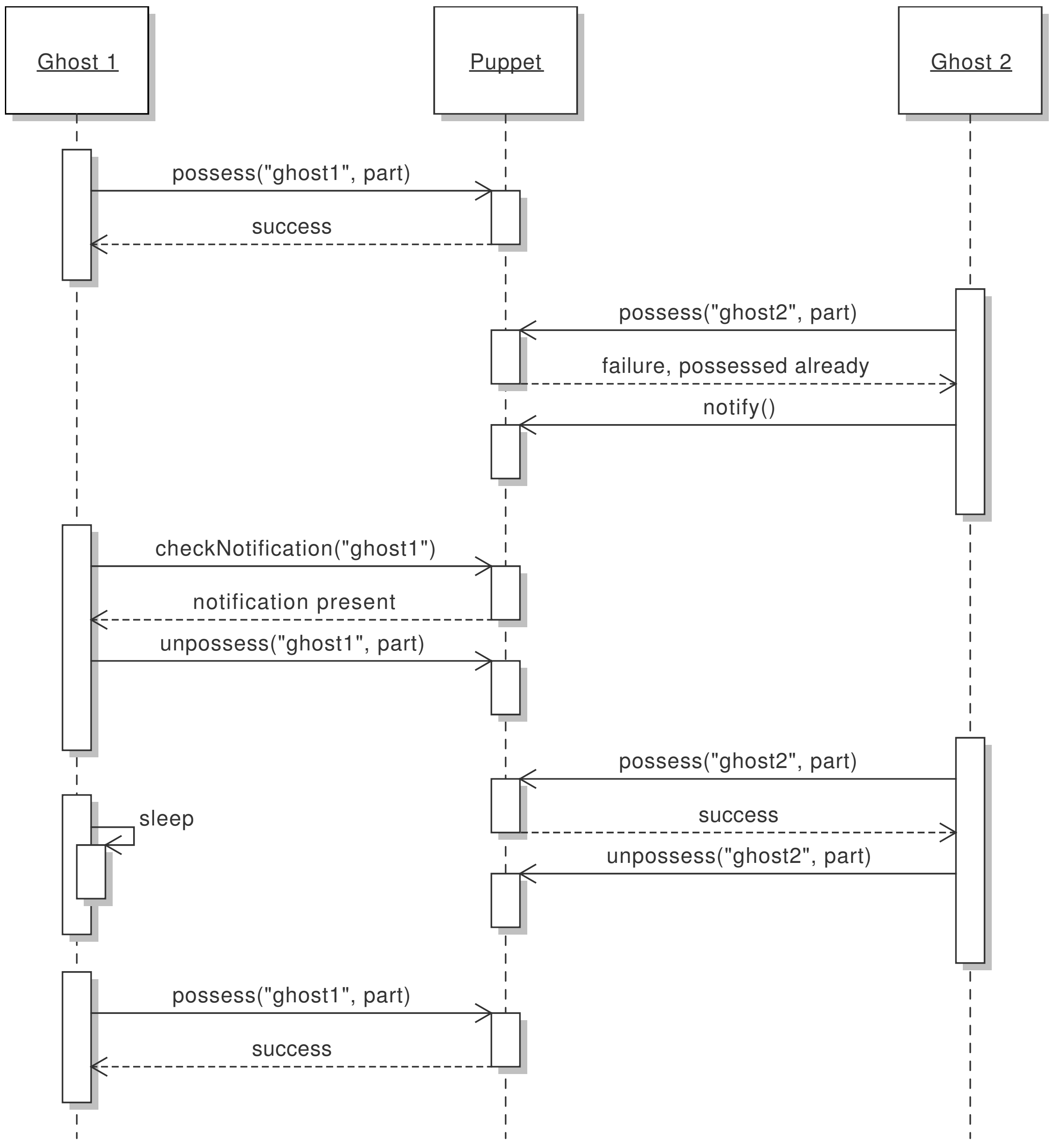


Figure 6.2: The puppet protocol

The information access goal is achieved both by the puppet and by the ghost. The API exposed by the puppet allows to interrogate the internal state even when not in control of the puppet. The puppet can also provide specialized APIs exposing the internal information from the animation system to the agents – such as position, orientation, animation state, etc.

The puppets act also as an abstraction layer with regards to the animation engine. A typical animation engine provides a set of low level animation primitives, such as object movement (translation, rotation), keyframe animation, inverse kinematics and procedural animation. These animation primitives frequently need to be combined together to achieve a meaningful animation of a character performing some action. The puppet is responsible for this task as well – it combines the primitive animations and builds high-level *actions* from them.

There are two mechanisms how the high level actions can be built:

- Regular actions – created by combination of the low level primitives described above.
- Generic actions – using object-specific scripts and data, typically defined by the smart objects.

The object-specific animations are typically created together with the smart object during the design process. The in-house developed animation engine VHD++ supports smart object use in the simulations, usually in places where a complex animation is necessary – typically object manipulation (as in figure 2.1). The object-specific animations are defined in the form of Python scripts using the object-specific data – such as points and direction vectors or grasping information – to perform a complex animation.

Figure 6.3 depicts a virtual human operating a jukebox. The goal for the virtual human is to turn the jukebox on by pushing a button on the front panel. Even for this simple task the animation is quite complex:

1. The virtual character has to approach the jukebox correctly (from the front, facing the panel, stop at the right distance). This is achieved by using the procedural walking engine available in VHD++. The approach direction and the stopping point are defined as attributes of the smart object and the script takes them into account to arrive at the jukebox in the correct way.
2. The button has to be pushed by the index finger, therefore another animation primitive is performed to put the hand into a "grasping" posture (hand in the fist form, index finger outstretched). The correct posture needed to achieve this is again part of the smart object and the script has to retrieve it before invoking the animation primitive.
3. Finally, the button has to be pushed down. This is achieved by inverse kinematics while the hand is held in the correct posture. The animation of the body and the arm is achieved procedurally using inverse kinematics, with the position of the buttons being part of the smart object definition.

It is obvious that a relatively simple action from the reasoning point of view – turning on a jukebox – is a complex animation problem. If such action had to be implemented by the puppet for each different object that the agent may encounter in the virtual environment, it would become intractable very quickly. The concept of generic actions is the solution here.

The generic action simply provides an interface to the ghost to invoke object-specific animations without having to know about them in advance. In the typical case, the ghost has to know only that it needs to power the object up, which is usually determined by object-specific planning described in section 5.4.

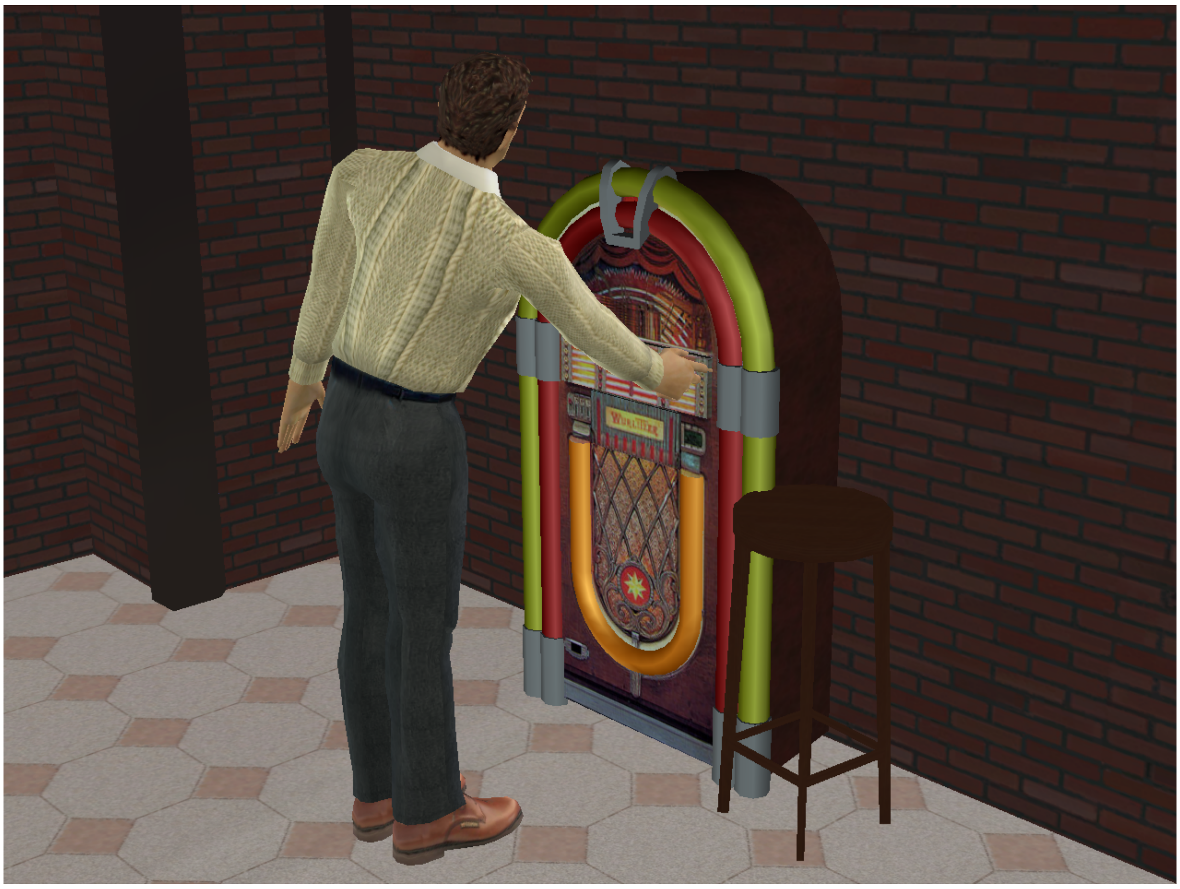


Figure 6.3: Smart object manipulation

The action is invoked with the 'power-up' and 'jukebox' as arguments. The real animation script is retrieved by the puppet from the smart object using these parameters as a key for looking up the necessary data and executed. The animation scripts are usually low level scripts intended to be executed by the animation engine, therefore they are uploaded using CORBA interfaces into the embedded Python interpreter in VHD++ for execution.

This scheme allows the puppet to contain only general actions such as locomotion and leave the object-specific issues to smart objects. In this way the complexity of the code and the development time is greatly reduced because the puppet developer has to care only about animation specific to the puppet, not to deal with object-specific issues.

The figure 6.4 contains the class diagram showing the relationships between the various components of the puppet interface. The base is the class `Puppet`, implementing the control sharing protocol. Every puppet needs to be a subclass of `Puppet` in order to be controllable by a ghost. Then there are several interfaces which define specific behavior, for example `machine` (for objects which can be powered up and down), `movableObject` (for objects which can be moved in some way, mostly interactively, e.g. by a gamepad) and the `genericAction` interface used to implement the generic actions.

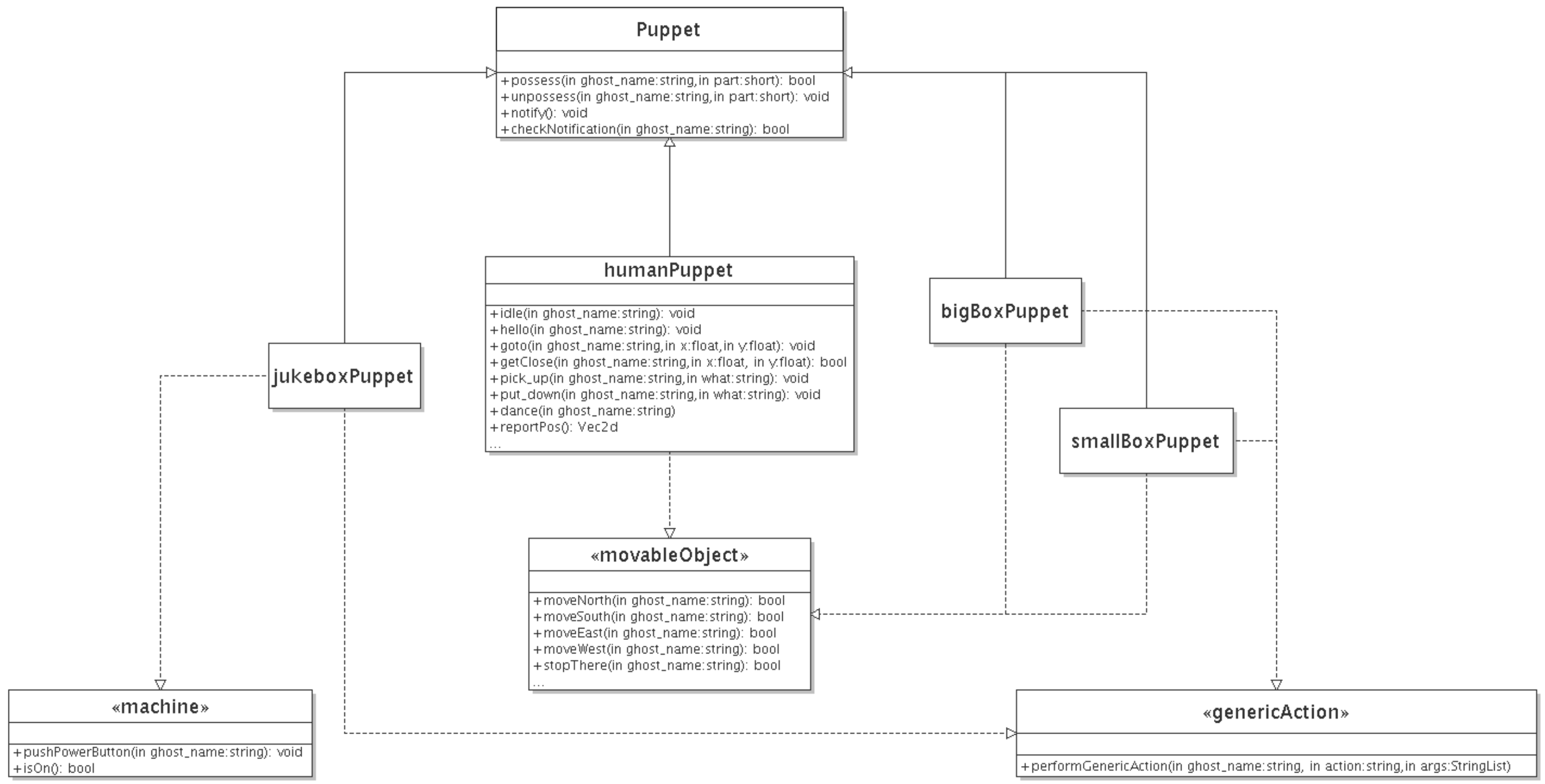

Puppet
+possess(in ghost_name:string,in part:short): bool
+unpossess(in ghost_name:string,in part:short): void
+notify(): void
+checkNotification(in ghost_name:string): bool
humanPuppet
+idle(in ghost_name:string): void
+hello(in ghost_name:string): void
+goto(in ghost_name:string,in x:float,in y:float): void
+getClose(in ghost_name:string,in x:float, in y:float): bool
+pick_up(in ghost_name:string,in what:string): void
+put_down(in ghost_name:string,in what:string): void
+dance(in ghost_name:string)
+reportPos(): Vec2d
...
bigBoxPuppet
smallBoxPuppet
jukeboxPuppet
«movableObject»
+moveNorth(in ghost_name:string): bool
+moveSouth(in ghost_name:string): bool
+moveEast(in ghost_name:string): bool
+moveWest(in ghost_name:string): bool
+stopThere(in ghost_name:string): bool
...
«machine»
+pushPowerButton(in ghost_name:string): void
+isOn(): bool
«genericAction»
+performGenericAction(in ghost_name:string, in action:string,in args:StringList)

Figure 6.4: Puppets

The class diagram is given only as an abbreviated example for space reasons. The real applications usually contain additional puppets for different active objects and use more interfaces as well, however the principle remains the same.

Ghost: The previous section mentioned the role of the ghost in the ghost & puppets framework as the core of the agent implementation. The main role of the ghost is to provide the "brains" for the agent. The ghost can be in control of one or several puppets, using their capabilities to express its actions in the virtual world.

As depicted in the figure 6.2, ghosts are not uniquely tied to the puppets. The control of a puppet can be shared or swapped between several ghosts using the described protocol.

An important role of the ghosts is to provide not only autonomous behavior to the virtual characters/objects and to monitor their state, but also to provide an interface for the human user to be able to interact with the simulation. Several control levels are possible and thanks to the dynamic control swapping, it is possible to use them interchangeably during the run time of the application. Such flexibility enables the user to use the application in the style which suits him best and to compensate the shortcomings of the individual control methods.

For the purposes of this thesis, three distinct control levels were implemented:

- Direct control – the user takes control of the puppet which becomes his *avatar*. A ghost implementing this control mode is typically acting as a translator from, for example, gamepad input by the user into the motion and actions of the virtual character in the virtual environment. The gamepad input is processed and the corresponding puppet functionality is invoked – for example to move one step in a certain direction.

  A more complex example of such direct control ghost will be described in the section 7.1, where a combination of an eye-tracking device with a gamepad was used to control virtual humans in a multi-modal manner.

  Another example is a multimodal control ghost in the "Virtual Guide" in section 7.2, where the user controls the actions of his avatar using the first-person view by a multi-modal combination of a handheld computer (PDA) and a special mat acting as directional control (similar to gamepad).

- Direct control by giving orders – the user controls the simulation by giving orders in some form and not by "driving" the virtual characters directly. The ghost is responsible for receiving the orders from the user in the preferred form (such as voice, typed text etc.) and transform them into *tasks* that the other agents are able to process. In the described framework the tasks are represented in the form of either an imperative or declarative task specification (see 3.1.3), usually a tuple.

  Ghost using this control level was implemented as a natural language interface for processing typed English in the section 7.3. It is also possible to interact in this way using the graphical user interface described in the section 7.4.

- Indirect control by proxy – whereas in the previous point the user was issuing orders directly to the agents, in this mode the interaction is performed via proxy. The orders are issued to a designated agent – a team leader – which then coordinates a group of others trying to execute the order. The user does not have to control each of the virtual characters himself. This interaction level is usually used for high-level tasks, where the problem is *delegated* to the team for solution and the user is not really interested in being involved in the process of solving it.

An example of such control arrangement is shown in section 7.6, where the user controls a simulated police force trying to maintain order in a city. The user commands the force by issuing orders to leaders, which in turn coordinate actions of the policemen.

6.2 Facilitator

The facilitator is a corner stone of the described framework. It has two main roles, as described in section 4.2.1:

- Central data storage (similar to a blackboard)
- Matching the incoming requests with the capabilities of the agents and delegating the tasks to the agents for execution.

The facilitator implementation fulfills these two roles by using few key concepts – unification, simplified solvables and minimal request tracking to keep the processing simple and fast. The facilitator is a central agent in the framework and almost every request passes through it, therefore the processing has to be very efficient to prevent it from becoming a bottleneck.

6.2.1 Implementation

The theoretical ideas behind the design of the facilitator were described in the section 4.2.2. It was implemented as a standalone agent using Python and omniORBpy library providing the CORBA binding for Python.

The facilitator processes the incoming requests according to the activity diagram in figure 6.5.

The facilitator agent offers three main entry points via its CORBA interface:

- Declaration of a new solvable. The solvable can be either a *data* solvable describing a state or a *procedural* solvable defining a capability of the agent declaring it. Data solvables declared by the agents define a global state of the world.

  The predicate defining a new solvable is presumed to be true, false predicates are not stored (closed world assumption). For efficiency reasons the internal knowledge base is maintained as hash table implemented using a Python dictionary.
- Un-declaration of an existing solvable. The solvable is removed from the internal knowledge base and the subsequent queries will assume it to be false.
- A delegation/state query request. Delegation and state query requests are invoked using the `solve(solvable, constraints, originator)` function.

The first two entry points are very simple – the corresponding solvable is either added or removed from the knowledge base. While adding the solvables, the CORBA reference to the calling agent is recorded as well, so that the declared solvable can be traced back to the declaring agent. This has no use for the data solvables, however it is very important for the procedural (capability) solvables. The recorded references are used to dispatch the delegated tasks to the agents declaring certain capability.

After modifying the knowledge base, the consistency is verified using a process of *axiom enforcement* to eliminate contradictions (inconsistencies).

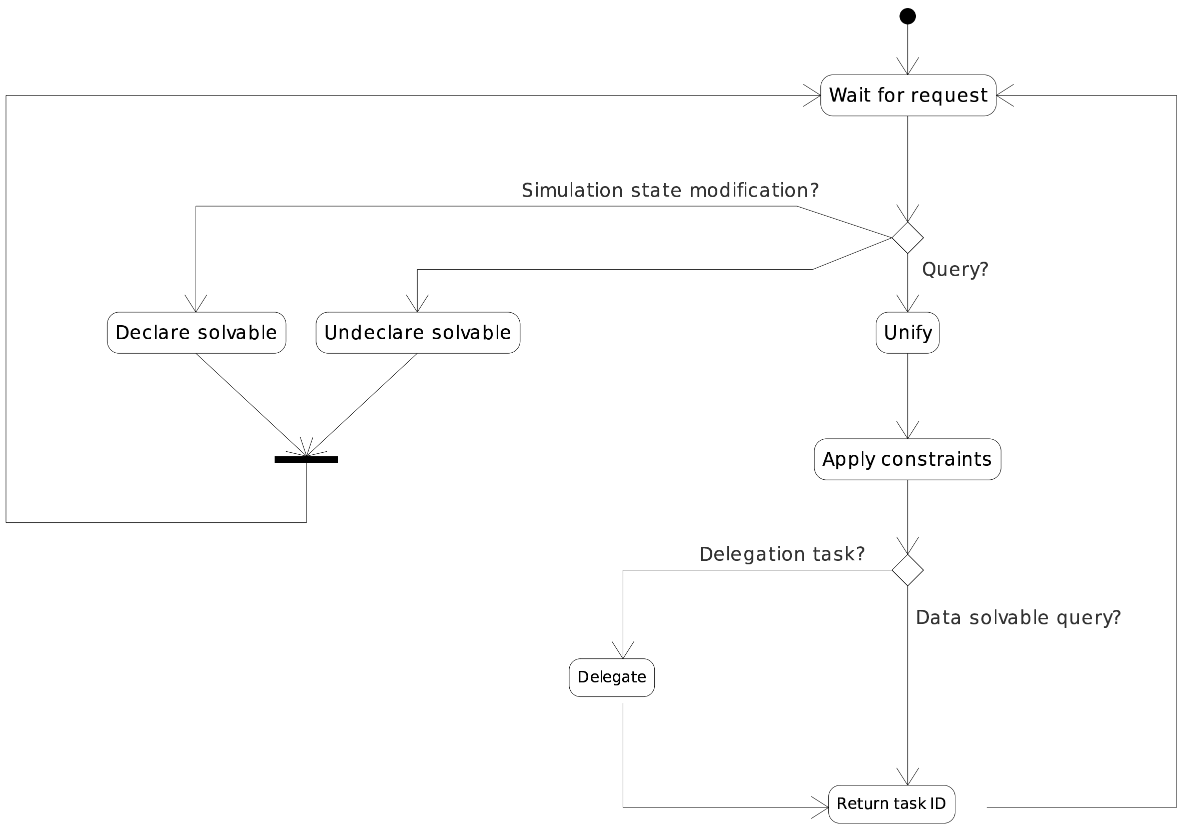


Figure 6.5: Activity diagram for the facilitator

The *axioms* are essentially rules expressed as implications. The premise is matched against the content of the knowledge base and for each match the consequent is added to the current state. The process will be described in detail in the section 6.3.7 in the context of an agent.

The delegation/query request is the most complex one. In fact, there are two distinct functionalities being provided by the same function – queries of the global state whether some solvable is defined (ergo, whether the corresponding predicate is true) and task delegation requests. These two tasks may seem unrelated, however there is deep interconnection between the two.

As seen in the figure 6.5, the solution process first tries to find all matches of the incoming solvable against the knowledge base. The matching is performed using unification, as described informally in section 4.2.3. The result of this step is the list of matches (unifications) with different variable bindings.

The unification can return three possible results:

- Result is `NONE`, meaning that no match was found and the incoming solvable does not unify with anything in the knowledge base. This result means that the queried solvable is either unknown to the facilitator (presumed to be false) or in case of a procedural solvable asking for a certain capability, the requested capability is not declared (and provided) by any agent.

- Result is an empty unification. This is a special case to deal with expressions starting with negation. The classic implementation of the unification algorithm uses a stream-based approach where the intermediate unifications are passed through a sequence of filtering nodes. Each filtering node corresponds to one logic connective of the original expression. In the case of the `not` connective, the original expression is negated and the negated expression is checked for matches using the bindings from the previous steps. If a match is found, the original expression does not match and the tested variable binding is rejected.

  This procedure assumes that there is some variable binding in the stream already. However, what may happen if the negation is the first sub-expression being evaluated is that there is nothing to return yet even though the expression matches (the negated one does not, since the stream is still empty). Returning `NONE` would signify that the unification failed, therefore a special *empty* unification which does not define any variable bindings was established for this purpose.

  This issue with `not` is an inherent problem stemming from filter-like implementation of the logical connectives. It is also described in [5].

- Result is a list of unifications (stream). In this case the searched information was found and is passed over to further processing.

The next step after determining the matching expressions from the knowledge base and their corresponding variable bindings is to apply the agent-specified constraints to narrow the working set of the unifications for further processing.

The current implementation of the facilitator is able to apply two kinds of constraints – numerical limit on the amount of results to be used and the capability constraint.

The number of the results returned/processed can be limited. The calling agent can specify the number of results to be returned using the `no_results` keyword. By default, all results (unifications) are used. In case a smaller amount is requested, a random pick is made by the facilitator.

The required capabilities of the agent the task will be delegated to can explicitly specified. This constraint is typically used while forming teams. It is expressed using the `has_capability` keyword and is usually used to tell the facilitator that the `recruit-help` call should recruit only agents having certain capability – for example being a driver or a barman. Without this constraint the team leader would have to recruit blindly and then reject the unsuitable agents from joining the team. Such approach is very inefficient because the facilitator has the required information already and can delegate the call to only relevant agents in the first place.

The final working set (stream) of unifications – in our case the list of matching expressions together with the corresponding variable bindings is passed to the final – delegation – stage. The facilitator iterates over all unifications in the list and in case that the match was for a procedural (capability) solvable, the task is delegated to its owner (the agent which declared it) together. Each request is assigned a task ID, which is passed along to both the *originator* of the request and to the *delegate*. The role of the ID is obvious – it allows the originator to match the incoming asynchronous replies from the delegates with the original request and corresponding reply from the facilitator.

In case that the delegation request from the facilitator to the agent fails (e.g. because the agent crashed and didn't remove its capabilities from the knowledge base of the facilitator), the error is handled by the facilitator. If it is still possible to satisfy the original request, the information about the failed agent is removed from the knowledge base and the processing continues. If it is not possible to satisfy the request (e.g. because the failed agent was the last one able to do so), error is reported to the delegating agent and the information about the offending agent is cleaned up.

Finally, in both cases that the query being solved was a delegation request or that it was a simple data solvable query, the unifications are returned to the request originator together with the assigned task ID as a return value of the facilitator's `solve()` call. For data solvable queries, the results are directly contained in the unifications being returned.

This facilitator design has proved as quite efficient and scalable, thanks to the limited processing the facilitator has to perform. The largest case study "Riot in the city" described in section 7.6 uses approx. 50 distinct agents driving more than 1000 virtual characters and vehicles on the screen which have to communicate via the facilitator in some manner in real time. On average, every agent declared ≈ 10 capabilities and ≈ 80 data solvables. Despite of this heavy utilization the facilitator was never a simulation bottleneck.

As an experiment, the most time consuming part of the processing which is the unification (especially in the "Riot in the city" case where the knowledge base contains many entries) was compiled into a C extension for Python using Pyrex[6]. The speedup was un-noticeable in practice, because the delegation and facilitator querying is only very small portion of the time spent by the agents while reasoning.

6.3 Agents

Software agents are the driving force behind any agent-based simulation. They are understood as autonomous software units interacting with their environment and frequently provide services to other agents or human users.

From implementation point of view, there are many relevant technologies for implementing agents, however for practical reasons outlined at the start of this chapter, the CORBA-based implementation was chosen. It is not unique choice, a similar CORBA-based systems are common in the industry. One such event-based system was described in [70].

6.3.1 Objectives

The objectives of the software agents present in the simulation are twofold:

- Source of autonomy in the simulation – the agents are the source of the autonomous decisions and actions in the multi-agent simulation. A typical example is a ghost agent driving a virtual character in the simulation.

- Provide services for others. Typically, agents providing services are not embodied and do not represent characters and objects of the virtual world – for example a "grid" agent or a "location" agent used in the case studies in the next chapter which provide geometry information and path planning service to the remaining agents. The facilitator agent belongs to this category as well.

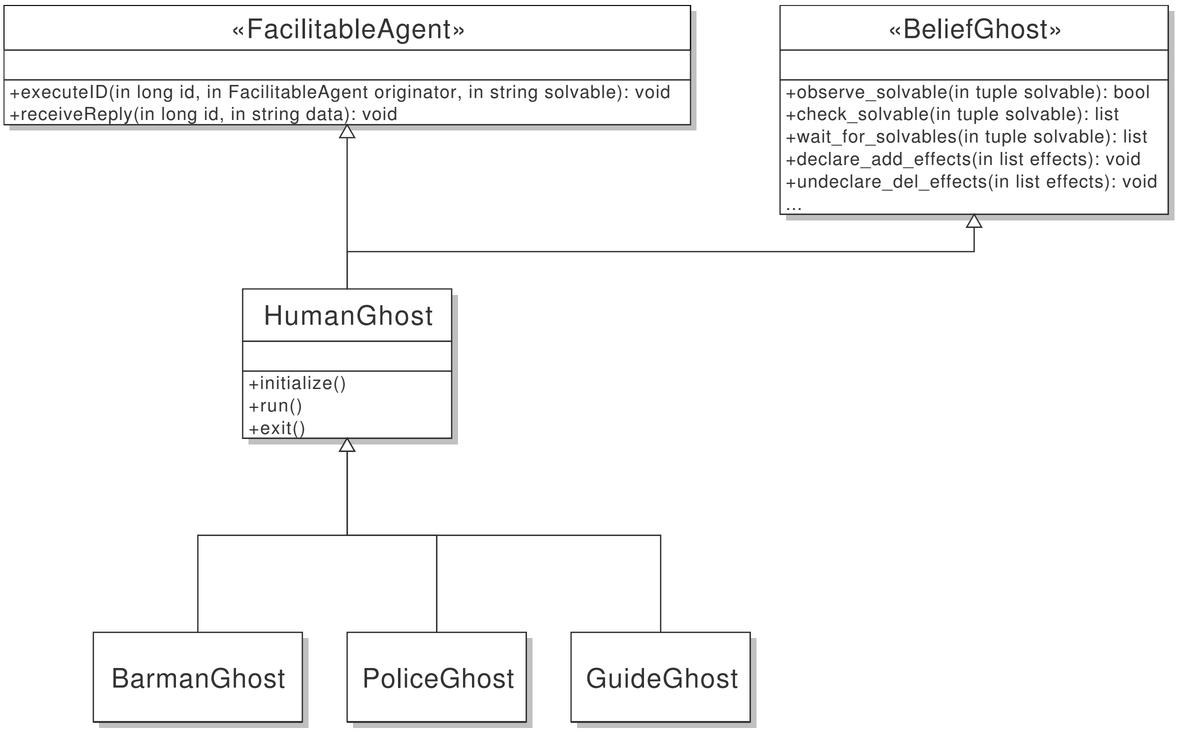
The ghost agents are a special type of a software agent. As written before, they are a part of the ghost & puppet framework allowing them to establish control over the puppets representing the virtual characters and objects in the virtual world. Consequently, the ghosts are embodied type of agents and the virtual characters are their avatars.

In the collaboration context, the ghost is the atomic component – the ghosts are the agents forming teams and working together to solve given problems. There are other agents playing auxiliary roles (such as the planner or the facilitator), however the ghosts, either autonomous or user-controlled, are the only agents truly participating in the intentional collaboration.

[6]`http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex/`

6.3.2 General design

Most of the agents are designed using the same pattern. There is a main loop, which controls the processing done by the agent. In most cases, the agent simply waits until there is a new task to perform in the internal job queue. When that happens, the loop is unblocked and various activities are performed – the task is processed, sensing is performed, some autonomous or idle activity is planned etc.



«FacilitableAgent»
+executeID(in long id, in FacilitableAgent originator, in string solvable): void
+receiveReply(in long id, in string data): void
«BeliefGhost»
+observe_solvable(in tuple solvable): bool
+check_solvable(in tuple solvable): list
+wait_for_solvables(in tuple solvable): list
+declare_add_effects(in list effects): void
+undeclare_del_effects(in list effects): void
...
HumanGhost
+initialize()
+run()
+exit()
BarmanGhost
PoliceGhost
GuideGhost

Figure 6.6: Autonomous agent class diagram

The tasks to do can come from two main sources – they can be either *external*, delegated from the facilitator or *internal*, created by the agent itself. The internal tasks are usually either autonomously generated actions (autonomous behavior) or as a special case – the idle task when there is nothing else to do.

The embodied agents driving human characters in the virtual environment are specializations of a generic `HumanGhost` class. This Python class implements a basic functionality common to all ghosts driving virtual characters, such as sensing, maintaining own set of beliefs, performing actions and interface to the facilitator.

The application-specific ghosts/agents fulfilling specialized roles in the scenario are derived from the `HumanGhost`, overriding the required functionality, such as adding new beliefs or special handling of certain situations.

The class diagram showing the relationships between the ghost classes is in figure 6.6. There is the already mentioned generic `HumanGhost` class, three examples of specialized ghosts which are used in the case studies described later and two interfaces.

The `FacilitableGhost` interface defines the methods the agent has to implement in order to be able to receive delegated tasks and results of the task delegated by it to other agents.

The `BeliefGhost` interface defines common functionality used by agents keeping their own beliefs about the virtual world. The belief-related functionality is not specific only to the ghosts, however it is most commonly used with them.

Figure 6.7 depicts the activity diagram of a typical ghost agent. Other, non-embodied, agents use a similar design, except that many activities that do not make sense in the non-embodied case (such as sensing) are not implemented. The reception of tasks from the facilitator and the replies to the delegated request is processed asynchronously using a separate thread of execution not depicted in the figure. The received data are queued, either in the job queue (delegated tasks) or reply queue (results of the delegated requests) where they will be picked up later.

The activities of each agent/ghost are driven by a main loop processing tasks (jobs) until the ghost is commanded to shut down. Jobs can consists either of single actions (delegated or normal) or a whole plan to be executed (e.g. as a result of delegated action which formed it during its execution).

Each iteration of the loop consists of several steps:

1. Sensing – the agent tests the state of the virtual environment.
2. Autonomous behavior – schedules actions to satisfy own desires, such as barman may want to keep the bar clean, customer wants to quench his thirst, etc.
3. Action execution – performs the action with the highest priority from the job queue. This is a complex step, because the correct actions has to be instantiated and the state of both agent's beliefs and the global world state in the facilitator has to be properly kept consistent. Moreover, the action execution has to be fail-safe – in case that the action can fail in an undesirable way, the failure has to be properly handled to prevent problems in the simulation. The action priority can be set by the scenario designer, however by default external (delegated) requests have higher priority than internally generated tasks.
4. Result reporting to the task originator – in case that the action was a delegated one, the result of the action (success/failure) has to be reported back to the originator of the request. In case that the task contained a plan to be executed, the result of the plan execution is reported. Result reporting is of utmost importance when team leader is coordinating several agents performing independent tasks.
5. Puppet notification check – at the end of each iteration of the main loop the ghost agent has to verify that nobody else is requesting control of the puppet. If the notification flag is raised, the ghost has to release control and try to reacquire it at the start of the next iteration.

Overall, the design of the ghost agent is a complex issue with many factors. The following subsections will focus on the specific issues concerning the parts of the design.

6.3.3 Beliefs

Agent's beliefs are the local knowledge the agent has about the virtual environment. The beliefs may (and frequently are) differing from the global state of the virtual environment kept by the facilitator. There are multiple reasons for this discrepancy, mainly the fact that the environment is dynamic with multiple agents which can change the state of the environment "behind the back" of the agent. Therefore an important part of the agent's workload is to keep its beliefs as consistent and up-to-date as possible.

To facilitate this goal, the agent's beliefs are stored in a Python dictionary (hash table) and can be modified in four different ways:

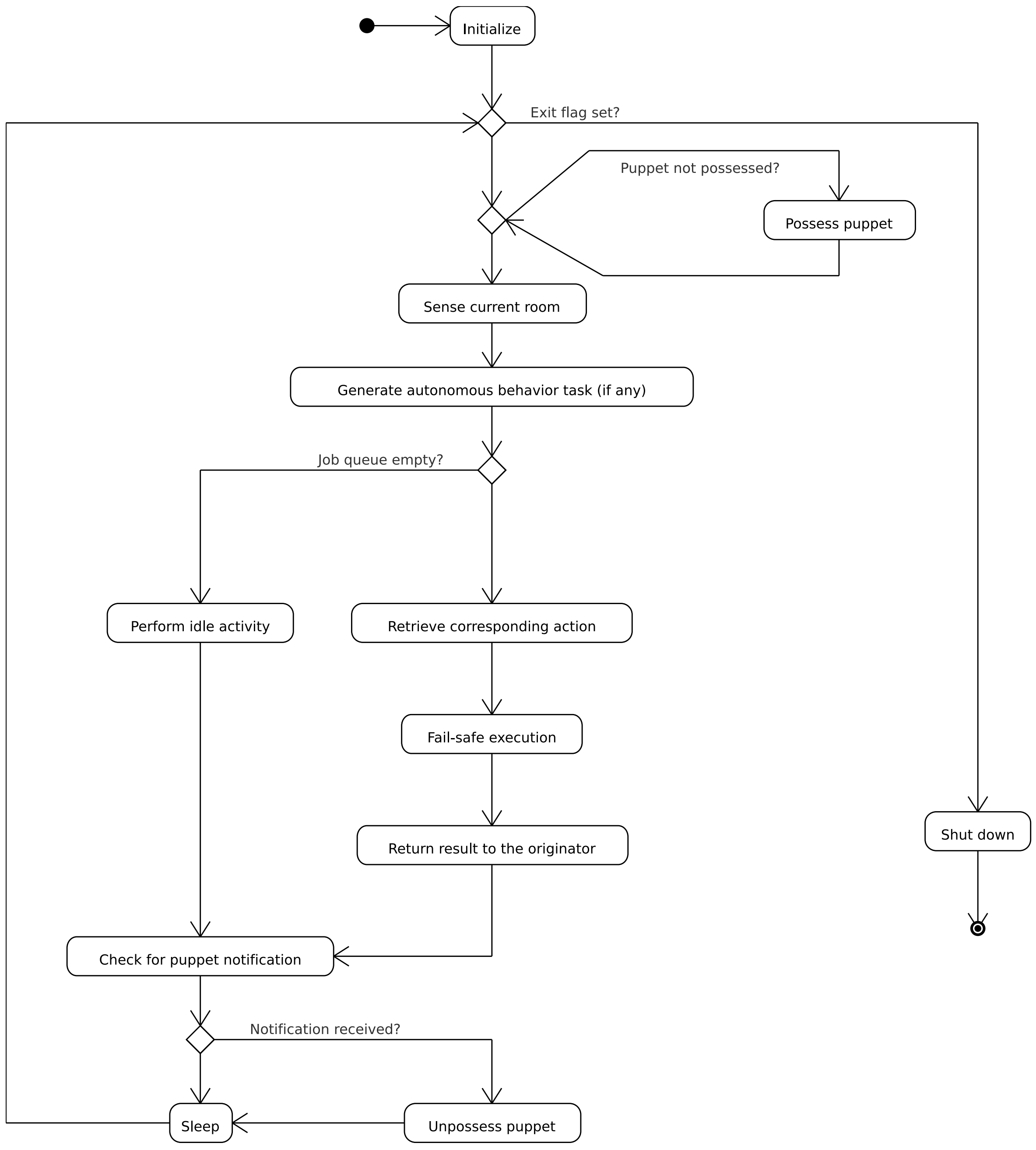


Figure 6.7: Ghost agent activity diagram

- Initialization – a priori knowledge about the environment and the agent itself is stored in the belief table at the beginning of the simulation.
- Action execution – after each action is executed, its effects are applied to both local and global (facilitator-maintained) state.
- Sensing – sensing updates the local beliefs with the perceived information from the environment around the agent.
- Axiom verification – after each change to the beliefs, the axioms have to be verified and discovered inconsistencies corrected.

All belief modification has to be done in a consistent way in order to avoid problems when the information is needed – such as when using the action planner. During the course of the simulation, this is normally ensured by the actions themselves.

There are two types of beliefs which are kept by the agent – symbolic information in the form of predicates (such as `(open door)`) and numeric fluents (such as `(= (THIRST) 40)`). Symbolic predicates are used to describe discrete information (such as door state), fluents express continuous values (team size, thirst etc.). From the agent's point of view, the fluents are represented simply by a table of numeric variables having symbolic names.

The agent's beliefs including both fluents and symbolic predicates form its *state*.

6.3.4 Sensing

Sensing is the agent's capability to perceive the surrounding environment and to learn the properties of it – to update its beliefs about the world.

There are many ways how to implement sensing capability, however in our case the sensing is meant to update the symbolic beliefs of the agent. Therefore, sensing should produce symbolic information. For simplicity, a very rudimentary sensing capability was implemented, where each room[7] of the virtual environment has a list of solvables to test for attached. These lists can be modified by the agents at run time and tell the agent to check whether the facts from the lists are defined or not while staying in the room corresponding to the list.

For example, a barman agent delivers a cup of coffee to a customer agent at a table. He signals the delivery by adding the solvable `(at coffee table)` to the list of solvables to test for the room "table". Once that happens, during the next observation cycle the customer agent will try to evaluate this expression in the context of the world state – it queries the facilitator to test whether the expression is true or not. In case that it is true, the expression is added to the agent's beliefs as well – he learned that there is a coffee there.

6.3.5 Actions

Actions are the most basic means for the agents to modify the state of the virtual environments. Usually, they have a corresponding animation associated (in case of the ghost agent the puppet is responsible for the animation), however there are actions without animation as well. Such actions are

[7]Room does not necessarily mean a room as in a house, it can be any space defined geometrically and tagged as such – such as the main square in the city. The purpose of rooms is to define topology of the world – which place is accessible from where – and to symbolically name the them.

used to modify the internal state of an agent, for example when performing an action plan created by the planner.

There are four distinct action types:

- Normal actions
- Internal actions
- Delegated actions
- Object-specific actions

Normal actions are used to perform animations – such as walking, pushing an object or operating a jukebox. Apart from the animation part, every action also changes the state of the virtual environment, e.g. the agent changes the declared position or an object has its state changed – such as a jukebox is turned on.

Internal actions are a special case without an associated animation. This typically also means that there is no interaction with the puppet and the ghost agent only changes its internal state – beliefs and potentially the global state in the facilitator as well (e.g. declares itself busy).

Delegated actions are used to implement the collaborative features as described in the chapter 4. They are the actions of the *delegate*, not the *delegating* agent. The delegated action is used when an another agent (a team leader) delegates the task via the facilitator to the agent. The incoming delegated task is inserted into the job queue and processed during the next iteration of the main loop.

Object-specific actions are retrieved during the process of object-specific planning from the smart objects. The dynamic nature and built-in introspection capabilities of the Python language are used to extract the relevant information from the smart objects and to make it available to the agent. Object-specific actions are tied to specific animations, such as opening a door, operating a jukebox etc.

All actions are implemented using a functor technique – every action is implemented as an independent executable class, providing information about:

- Action's Name
- Arguments
- Preconditions
- Effects
- Auxiliary information (used fluents, predicates, etc.)

This information corresponds to the information required for describing the task semantics using the *operators*[8] from the section 3.1.4 and is in fact used for generating the data necessary for the action planner as well.

Every action class is providing methods to verify its preconditions with regards to the agent-provided state and to execute it. Execution of each action is usually a four step process:

[8]There is an unfortunate terminology clash here, because strictly speaking, "action" in the symbolic sense means an instance of an operator, whereas here the term "action" is used to describe the operator itself together with the animation information attached, as customary when speaking about agents. In most cases it is clear from the context which meaning of "action" is intended.

1. Preconditions are verified using the agent-provided state. In case that the preconditions are not satisfied, the execution is aborted and an action failure is signaled.

   The precondition verification itself is implemented using unification over the current state of the agent (its beliefs), not the global state stored in the facilitator, in order to preserve the partial observability principle, as described in the section 3.1.2.

2. The action is performed. Path planning may be requested, additional information may be retrieved from the other agents in the system – the facilitator, location agent providing geometrical information or others. Finally the animation primitives provided by the puppet are called in order to perform the desired action, such as walking, object manipulation, etc.

3. The effects of the action are applied to the agent's and world's state. The declared add-effects are added to both facilitator and agent's beliefs and the del-effects are removed. In case that a failure occurred during the action execution (exception from the puppet, animation problem, collision, etc.) the ghost agent is responsible for recovering to a meaningful state.

   Each action is implemented in such way that should a failure occur, the agent and the world are left in a consistent state – if possible, the failed action is undone and the effects reverted. If such solution is not possible or not desirable, the state has to be updated to match the current situation – for example if the `move` action failed during movement towards the goal because the visualization engine reported collision with an unexpected obstacle, the action has probably moved the agent out of the initial position. Therefore the state will be updated to reflect the current position of the agent – which is probably different from both the starting position and the goal position.

   Such error recovery requires reasonable granularity of the actions. If the actions are too high level, the recovery will be very complex, because lot of the state has changed during the execution. If the actions are too low level (e.g. mirroring the animation primitives provided by the visualization engine), the recovery will be easier because the impact of each action will be more limited. However the performance will suffer because even a simple task will require very long and detailed plans. This is again a domain-specific issue which has to be carefully balanced by the scenario designer.

4. The results of an action are reported to the caller (the ghost agent) as a return value of the action functor. Both success and failure are reported, in the failure case the reason for a failure is provided as well (e.g. obstacle, no path to goal, unknown object to manipulate, etc.). The reports are provided in a form of a pair, where the first element is either `success` or `error` and the second element contains the reason for the failure, such as `nopath` (no path was found), `precond` (precondition not satisfied) and others. For example, a path planning failure is reported as `(error nopath)`. The ontology is defined in a semi-formal way as dictionary reference for the scenario designer.

Actions are implemented separately from agents, they are not a fixed part of the agent code. To increase flexibility and to allow easy reuse of common actions between the scenarios (such as `move` or `pick-up`), agents discover and load all actions except the object-specific ones during the initialization dynamically, using the dynamic compilation capabilities of Python.

The delegated actions are a special case. Their implementation is usually done using the planner, which is given the effects of the delegated action as a goal state. The rationale for this is that the delegated action can express a high-level goal (such as "move to a place X") without regard on the constraints placed on the agent (e.g. not being able to move through obstacles). The planner is used to solve this problem by creating an action plan to achieve the desired state.

The object-specific actions are the second special case – they are loaded whenever the agent has to solve a problem using a planner. In that case, the smart objects potentially relevant to the planning goal are identified using a simple goal analysis heuristics (see section 6.4.1 for implementation details) and scanned whether they contain any object-specific actions. In the affirmative case, the actions are downloaded and dynamically compiled, in the same manner as the other actions during the initialization. This capability enables the agent to "learn" new skills at run-time, by investigating the smart objects.

6.3.6 Application of the effects

After each action is performed, it is necessary to update both the local state of the agent (its beliefs) and the global state tracked by the facilitator to reflect the changed reality of the virtual environment. To achieve this task, the effects declared inside of each action have to be applied to both the local and global state.

There are two different kinds of effects which have to be distinguished:

- Local effects – effects need to be applied only to the state of the agent executing the action and to the facilitator. No other agents are involved.
- Remote effects – effects need to be applied both to the local agent, to some remote agent and to the facilitator.
- Induced effects – effects forced by the state change of the agent while forming/dissolving teams.

The local effects are the simpler case – each action apart from the delegated ones has a set of effects declared, which are applied to the current state (the add-effects are added to the table of beliefs, the del-effects are removed from the same), both for the local (agent's) state and to the global (facilitator's) state.

The remote effects are more complicated. In traditional systems which do not use task delegation there is no need to consider this case. However, in case that a team leader agent delegates a task to a teammate, both agents have to apply the results of the action – the team member performing the task applies all the effects from all actions which he performed during the execution of the delegated action and the team leader applies the effects declared by the delegated action to account for the state change made by the team member. It is obvious, that two sets of effects may be different, typically because the team member has to perform work which the team leader does not know about – there are hidden side effects.

The side effects are a potential cause for inconsistencies in the beliefs of the agents, let's consider this simple example:

1. The team leader asks agent Gino to open a door for him to pass through.
2. Gino needs to approach the door in order to open it and obstructs it in the process.
3. After the door is opened, the team leader has only information that the door is open and the (outdated) information that he is able to pass through the door.
4. The team leader fails to pass through the door because there is an obstacle – Gino.

The problem in this example stems from the limited information available when a delegated action is performed. The delegating agent cannot make any assumptions on what the delegate will do during the course of performing the delegated task, leading to inconsistency. The agents re-sense their environment before executing the next action and the impact of this problem is reduced (not fully eliminated – the agent may not sense ("see") all changed performed by the side effects).

Induced effects are a special case when dealing with the protoagents and team creation by the team leader. When a new team member is recruited, all symbolic information associated with the corresponding protoagent has to be applied to the new agent as well. For example the team leader has protoagent `A1` designated as a barman by having a predicate `(barman A1)` in his beliefs. After recruiting a new team member Gino to take up the role of this protoagent, the information that Gino is the barman has to be inserted into team leader's beliefs – the effect of Gino being declared as an agent in the team leader's beliefs *induces* the effect that he is a barman. A similar process is performed when the team is being disbanded – the additional information about the disbanded agent has to be removed from team leader's beliefs.

6.3.7 Axiom enforcement

It is important to ensure that the consistency of the world representation used by the agents is maintained. The previous sections described the possible problems already and how they can be addressed. However, from a practical point of view, it is beneficial to have a possibility to check and strictly enforce some fixed constraints – such as that the agent can be only in one place at a time or that in order to be near to some object, the agent has to be in the same room (semantically) as the object is. Such constraints are useful from two points of view:

- Constraints help discover bugs in the action implementation – if a constraint is violated, there is most likely a hidden problem somewhere in the action effects that needs to be fixed, because it may lead to more serious issues.

- It is not possible to check for every possible dependency between two (or several) actions. For example if an agent is near a jukebox, expressed by `(near gino jukebox)`, moving the agent away from the room with the jukebox should delete also this predicate (the agent is not near the jukebox anymore). However, that requires that the effects of the `move` action explicitly take this issue into account. There could be many such predicates that need to be removed from the agent's beliefs when an action is performed and the problem grows to intractable state rather quickly. The solution is to let the constraint checker to handle such "common sense" issues.

Such constraints – or axioms – are tested and enforced after performing any action that can modify the state of an agent. A similar step is performed by the facilitator as well, whenever a data solvable is added to or removed from the global state.

Mathematically, axioms are formulas which are universally valid in the given domain. In the presented case, the notion of axiom is being used in a slightly different context – an axiom is a rule, which enforces an universally valid concept, such as that an agent can be only in a single place at any given time.

The axioms are expressed, as described in the section 3.1.1, in the form of a logical implication $A \Rightarrow B$, where A and B are predicates. From the implementation point of view, the axioms are tested and applied as a two step rule:

1. The premise of the axiom is matched against the current state of the agent (either the beliefs of a ghost agent or the state of the facilitator).

2. If a match is found, the consequent is applied to the state (unless it was there already).

From the formal point of view, it is possible to treat these axioms as a special STRIP-s like operators, by rewriting an axiom $A(\overrightarrow{x}) \Rightarrow B(\overrightarrow{x})$ as an operator with arguments $\overrightarrow{x}$, precondition $A(\overrightarrow{x})$ and effects $B(\overrightarrow{x})$ (similar approach as used to support derived predicates in planners, see Thiébaux et al. [107]). Then it is possible to use situation calculus to describe formal semantics of the axiom application on the world state, as described in the section 3.1.4 already.

The derivation process is known as *modus ponens*. Let A and B be predicates. Predicate A holds, because a match was found in the state of the agent. $A \Rightarrow B$ holds as well, because it is an axiom defined for the scenario. By applying modus ponens, it can be inferred that B holds as well – therefore it needs to be applied to the local state.

It was mentioned in the section 3.1.1 that the universal quantification is implicit for the axioms, due to the way how the expression is evaluated. In the described case, the matching is performed by unification which always returns all possible matches – essentially simulating the effect of the universal quantification.

6.4 Action Planning

Action planning allows the agents to solve complex problems and to collaborate using the delegated actions. There are many possible ways how to approach this problem, however for the purposes of this thesis a STRIPS-like system was selected.

The implemented framework is using two different action planners in the form of the planning agents – Sensory Graphplan and Metric Fast Forward planner described in the sections 2.5.2 and 2.5.2. They are both STRIPS-like planners, however each of them has different properties and is better fit for different scenario.

6.4.1 Planning by agents

The agents invoke the planning agents to create plans using one of the two available planners to solve the problems given to them in the form of goals. They use two special actions for this purpose which do not have any preconditions nor effects declared. These two special actions are `plan-and-execute` and `execute-plan`. The plans are built using the local beliefs and capabilities of the agent invoking the planning, the required data (state & operators) are passed as a part of the problem specification to the planning agent.

Whenever the user or another agent requires an agent to solve for a certain goal, the task is delegated to the agent in the declarative form as an argument of the `plan-and-execute` action, as described in the section 3.1.3. The facilitator matches the action against the agents offering it and dispatches it to the correct recipient. This process was depicted in the figure 5.1.

Upon receiving this special task, the agent starts executing the planning process, as shown in the activity diagram in figure 6.8. The processing consists of multiple steps:

1. Object analysis – in case that smart objects are used, they are examined for relevant object-specific actions and properties. The relevant smart objects are identified using a simple goal analysis – every object mentioned in the goal specification is tested whether it is a smart object

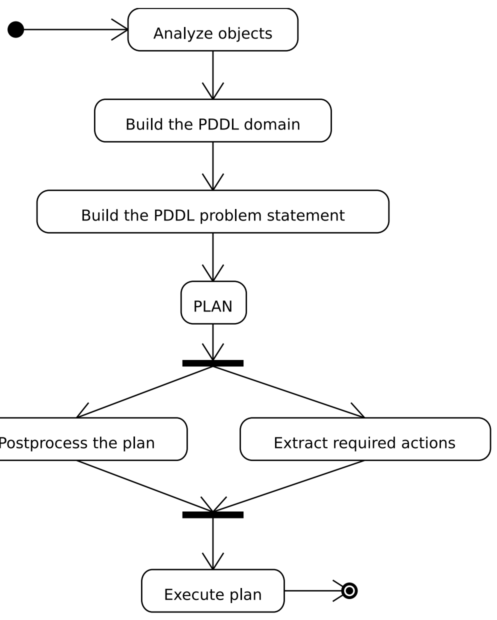


Figure 6.8: Activity diagram of the plan-and-execute action

and in the positive case it is examined for object-specific actions and properties which are then added to the beliefs of the agent. Additionally, the objects from the sensing list for the current room are examined in the same way.

The object analysis heuristics is not perfect, with certain goals it is easily possible to miss the relevant object. Problematic goals are such which do not explicitly mention the object needed to perform the action – for example goal `(light-on living-room)` does not mention the object `light-switch` needed to turn the lights on and the agent has to discover the object by other means – such by sensing the room and examining every object it has found.

However, the described heuristics works well enough for practical purposes because usually the object to be interacted with is in the same room as the agent (and therefore likely to be on the sensing list) or it is frequently specified in the goal statement.

2. The PDDL domain is built – this step constructs the domain for the planner consisting of declarations of the operators and possible predicates. These data are retrieved from the action functors of the agent.

3. The PDDL problem statement is built – in this step the initial and goal state are created for the planner, consisting of the lists of predicates defining the initial state and the goal expression.

4. Planning is performed – the agents do not plan themselves, the planning task in the form of the PDDL domain and problem statement is sent to a centralized planning agent which acts as a front-end to the actual planner. The benefit of such centralized approach is that the planner can be easily offloaded to a specialized machine (such as a high performance cluster) and shared among many agents.

5. Plan post-processing – during the post processing stage, the plan is simplified and cleaned up. Actions which were split for the cost adjusting purposes (as described in section 5.3) are merged into a single action again and the now unneeded intermediate steps are removed.

6. Required action extraction – in order to be able to recruit team members which are able to execute the created plan, each protoagent has a list of actions it has to be capable of performing attached. The list is built by extracting the actions the protoagent has to perform from the plan and it will be used during the plan execution to constrain the team member recruitment to only suitable agents. For example, if the plan calls for protoagent A to serve drinks as barman, only agents having this capability will be recruited for this protoagent.

7. The plan is executed using the `execute-plan` action.

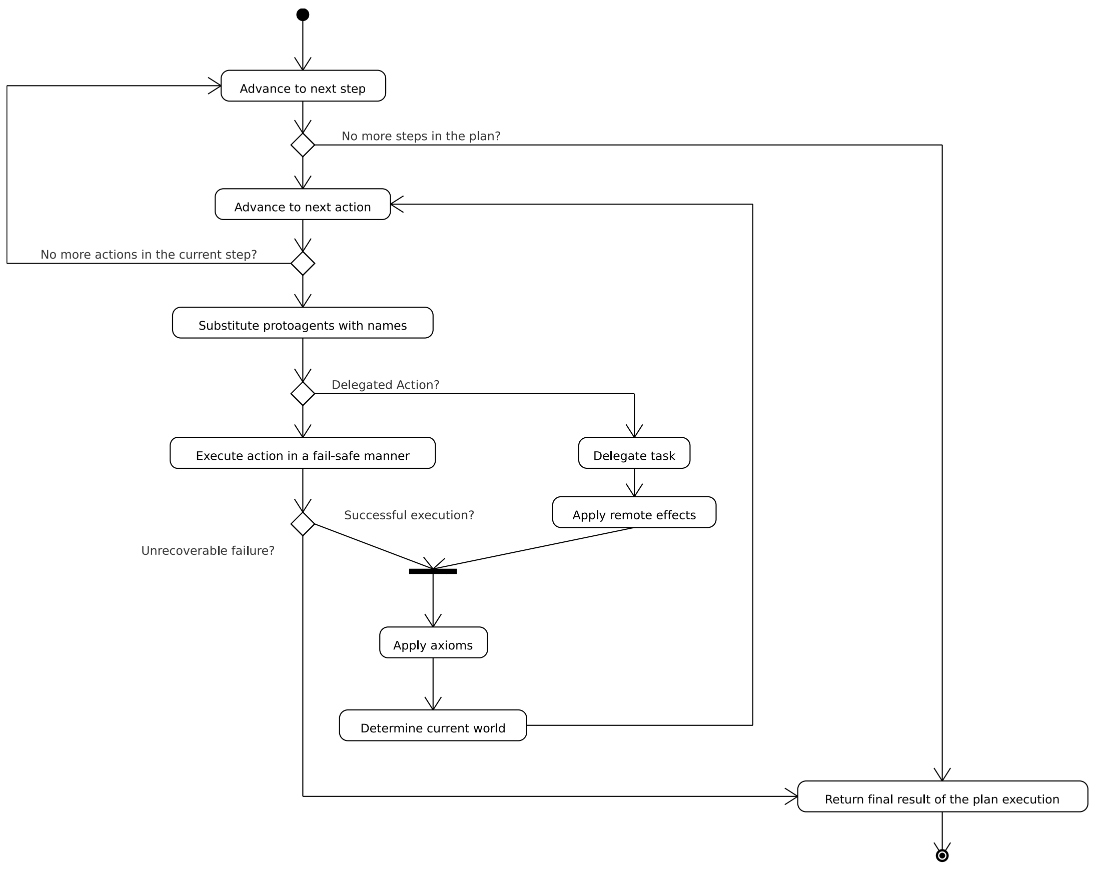

Advance to next step
No more steps in the plan?
Advance to next action
No more actions in the current step?
Substitute protoagents with names
Delegated Action?
Execute action in a fail-safe manner
Delegate task
Successful execution?
Apply remote effects
Unrecoverable failure?
Apply axioms
Determine current world
Return final result of the plan execution

Figure 6.9: Activity diagram of execute-plan action

Execution of the plan is shown in figure 6.9. It is again a multi-step process, however rather simple. The algorithm assumes, that the plan is a partial order (parallel) plan coming together with the corresponding world description from the SGP planner. In case that the M-FF planner is used, every planning step contains only a single action and there is only a single world.

In the case of total order plans, where parallel execution is desired, one additional pre-processing step is added. All recruiting actions are moved to the beginning of the plan, and then the total order plan is iteratively collapsed into several planning steps where each step contains multiple independent actions to be executed in parallel. The collapsing process keeps the plan partially ordered, the actions can be only either collapsed into a single planning step or a new step has to created. It is not allowed to invert the order of the actions.

The decision whether it is possible to collapse an action in the current planning step or whether a new planning step has to be created depends on whether the action is mutually exclusive with some action present in the planning step already. If the action is mutex, it cannot be collapsed into the current planning step and a new one has to be created.

To determine whether two actions are mutex or not is possible using the same technique the original Graphplan planner uses – the actions are tested, whether their effects are conflicting (such as producing both A and $\neg A$ in the current state) or whether they have competing needs (one action undoes the precondition of the other). These tests are described in detail in [10].

The collapsing of the plan is an action which is not always possible. The problems are caused typically by the tendency of the total order planner to emit an action into the plan only at the moment when it is actually required, thus producing chains of sequentially dependent actions. Such plans are impossible to convert to partial-order plans. This fact is also a reason why the recruiting step has to be moved to the start of the plan, to “free” the following actions on the protoagents.

The collapsed plan is then executed in a slightly different manner – all delegated actions present on the same level are delegated to the facilitator at once (not sequentially) and the execution proceeds to the next planning step only after all the actions present in the current planning step are completed.

The “Substitute protoagents with names” step in the figure 6.9 is used to replace the protoagents used by the planner with the agents which were recruited for them, because the protoagents are specific to each agent – protoagent A1 may denote different team member for agent Gino than A1 for agent Carlo.

6.5 Fail-safe execution

The described multi-agent system enables complex simulation, however there are many possible failure modes. Apart of the ones which are intrinsic to distributed design of the framework (connection failures, networking issues, etc.), there are problems which stem from the dynamic nature of the simulation and limited agent’s knowledge about its environment – such as problems with side effects of delegated actions or action failures because of stale information.

It is important to ensure that the agent will remain in a consistent and usable state regardless to what happens during the action execution. This requirement ensures basic robustness of the system (i.e. one failed action will not block the whole scenario). To provide this capability a concept of *safe execution* was created.

An action is said to be safely executed if it is executed inside of a specially crafted sandbox – a wrapper which tests whether there is an action-specific *failure handler* provided and executes the action using this failure handler.

The failure handlers are special functions registered by the application developer for any action where the default behavior (returning the failure code) is not sufficient or desirable. The failure handler can, for example, arrange for re-planning in case of a failure or ensure that the action will be retried until it succeeds instead of failing. They are completely separate functions from the action functors. The main reason is flexibility – different agents may have different needs how to handle failures.

Apart from failure handlers, the action execution safety is ensured by using atomic state updates in both facilitator and local beliefs and using the axiom enforcement. Each action can apply its effects if and only if the action was successful. In case of failure, the state recovery needs to be done – both by the action code and also by the potentially attached failure handler.

All these techniques try to ensure that the framework is kept in a consistent state, regardless of problems that may occur due to the limited knowledge of the agents and the side effects of the delegated actions.

Of course, some failures still *do* occur, for example if an impossible task was assigned to an agent or if the situation changed in an unexpected way. However, such failures are realistic. They occur in the real world as well and it is impossible (and probably undesirable too) to try to prevent them.

6.6 Summary

This chapter presented the design and implementation of the proposed collaboration framework for autonomous agents based on the task delegation and action planning. Autonomous agents are using the framework to collaboratively solve the tasks assigned by the user or their respective team leaders.

The problem solving process is a complex task consisting of several important steps:

1. The task has to be formally specified, either in an imperative or declarative manner.
2. The task is delegated to the facilitator agent by either an autonomous agent or by an agent representing the human user.
3. The facilitator has to find agents which are capable of performing the requested task and to dispatch the specification to them.
4. The receiving agent(s) either executes the task directly (imperative specification) or use the action planner in the form of a planning agent to find a solution of the task (declarative specification).
5. If necessary, the agent will build a team of agents with the required capabilities to be able to execute the created plan. The agent becomes a leader of a team of agents working together.
6. The team members are told to solve sub-problems, using delegated actions. The delegated actions could lead to another round of action planning by the delegates, naturally implementing a hierarchical planning scheme.
7. The team leader oversees the progress of the sub-goals and coordinates further actions as needed until the problem is solved.

Consistency of the logical representation of the simulation environment is of great importance. Several mechanisms were presented which are used to improve the situation – sensing, effect application and axiom enforcement.

The virtual characters and objects in the virtual environment are controlled by the ghosts & puppets framework. The puppets implement the low level basic animation features (such as walking or keyframe animation). The ghost concept is an agent abstraction for asserting control over a puppet ("possessing" it). The ghost agents are used to implement the majority of the autonomous agents controlling virtual characters in the scenario.

The user is represented by a special ghost(s) which translate the user's input into task specifications usable by the described framework. The user can work on several levels, using either direct or indirect control over the scenario and adapt his work style to his needs and preferences.

There are multiple special, non-embodied agents performing auxiliary roles in the simulation. The most notable one is the facilitator, which is used to maintain the state of the simulation and to match the offered services with the requested tasks. Facilitator uses the unification algorithm to evaluate queries about the state of the world and also to delegate the requested tasks to agents able to perform them.

Finally, the implementation of object-specific actions was introduced, allowing the scenario designer to create "pre-packaged" smart objects which are readily usable by any agent supporting them, greatly enhancing the reusability of the data between the scenarios.

Chapter 7

Case studies

The processes and frameworks developed during the course of this thesis and described in the previous chapters were tested and evaluated using multiple case studies. These case studies demonstrate various capabilities and properties of the described framework in real applications.

Each case study will be examined from several points of view:

- User input (interaction mode, how is the user involved)
- Collaboration style (ad hoc tasks, problem solving, team formed/no team, etc.)
- Evaluation

The user's involvement in the simulation will be examined – how the user interacts with the scenario and which interaction styles are available. Some scenarios use direct interaction, where the user is controlling the simulation by direct input (such as by using joystick). Other case studies use indirect input or a combined setup with both direct and indirect control.

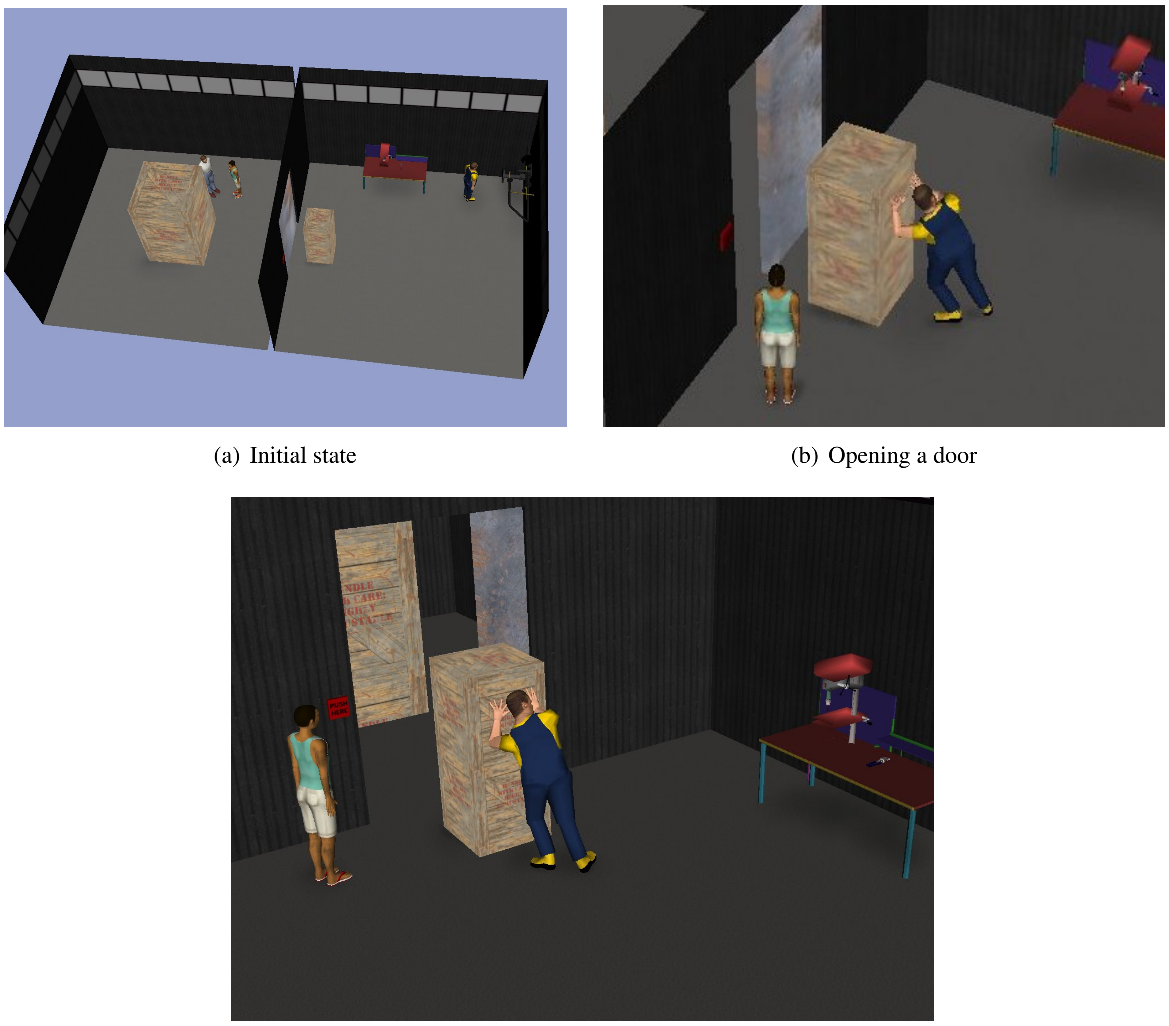
The collaboration style will be examined as well. The user can collaborate with the other agents either in an ad hoc way – by sending direct orders (imperative task specification) or in a more organized way – by forming a team and using the problem solving capabilities of the agents.

Finally, for each case study there will be a short evaluation summarizing the observations made during the development and use of the scenario, mainly from a usability point of view.

7.1 "Box world"

The "Box world" scenario was designed mainly as an initial test bed for the development of the collaboration and problem solving framework. The scenario contains two (or more) virtual characters, several crates and two places (rooms) separated by a door. The goal for the virtual characters is to move the crates from one room to another. There is a constraint imposed on the virtual characters that the agent moving a crate is unable to open the door (is encumbered). Alternatively, the crate to be moved is too heavy/large for a single character to move. Therefore, two or more virtual characters have to work together to solve the given problem.

Two variants of this scenario exist – the original one with a warehouse (fig. 7.1) and a more recent one, taking place in front of a virtual museum (fig. 7.2). The difference is mainly in how the crates are handled – the museum variant uses object-specific planning and object-specific actions to move the heavy large crate by coordinating two virtual humans, whereas the older scenario with the warehouse used a "hardwired" approach with fixed actions.



(a) Initial state (b) Opening a door

(c) Moving the crate

Figure 7.1: Warehouse scenario

The user is represented in the virtual environment as two ghost agents translating his input to orders for the virtual characters. There are all three interaction levels possible:

- Use the gamepad ghost to take control of one of the virtual characters or even crates and move the crates directly.
- Collaborate with the other virtual characters directly by giving imperative orders (task specifications). The user is able to either move the crates and have the other autonomous agents open the door for him or vice versa.
- Work on the high level by asking the agents to solve the problem themselves. In this variant the user utilizes a simple GUI to type in the correct task specification directly and delegate it via the facilitator to the agents.

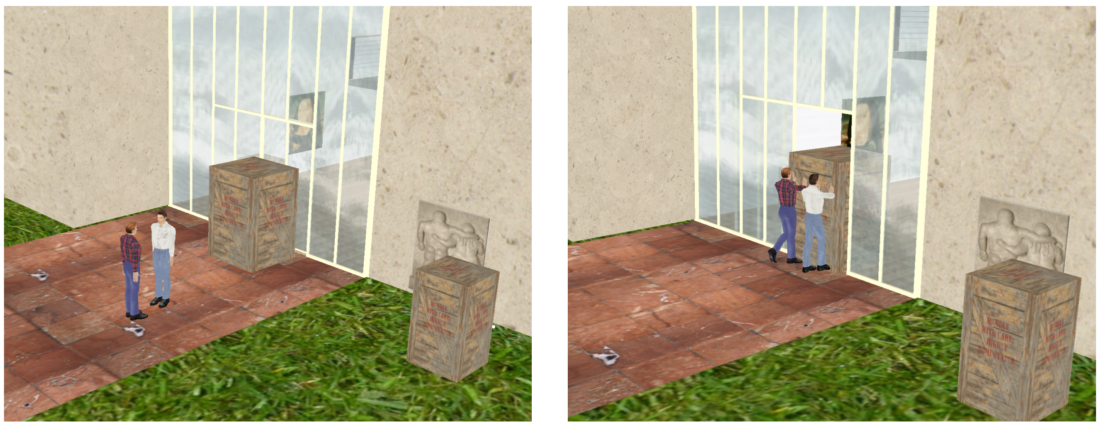

(a) Initial state (b) Moving the crate

Figure 7.2: Museum scenario

Apart of testing the task delegation, the ghost & puppet framework and the basic collaboration concepts, the “box world” scenario was used also for development of the object-specific planning. Using the object specific actions embedded into the smart object representing the large crate, the agent is able to discover that the crate is too heavy and that two people are necessary to move it. The agent is then able to recruit a teammate to help him to move the box.

The box pushing animation was implemented using inverse kinematics – the box is translated in sync with the forward motion of the agents and their hands are held in position using the IK. The full description of the approach can be found in [1, 2].

7.1.1 Eye tracker experiment

As an experiment, an alternative control ghost was implemented and tested in the “box world” scenario. The user is able to control the virtual characters using the VisionTrak eye tracking system. The system was published in [13].

The system was used mainly to verify the ability to process real-time data from the eye tracking and magnetic tracking systems using the multi-agent framework and to evaluate the utility of eye tracking for interaction and monitoring.

The user uses a multimodal setup with the eye tracker and a gamepad to drive the virtual characters in the warehouse, as seen in fig. 7.3.

The eye tracking system was used to implement a “go there!” metaphor. The system has two modes:

- Indirect, order mode – the user picks an object or place by gaze and asks the virtual character to approach the picked target by pushing a button on the gamepad.
- Direct mode – the virtual character follows the gaze direction.

The selection of the virtual character to control is achieved by gaze and the user’s ghost agent will take control of the puppet driving the virtual character when the selection is confirmed by pushing a button on the gamepad.

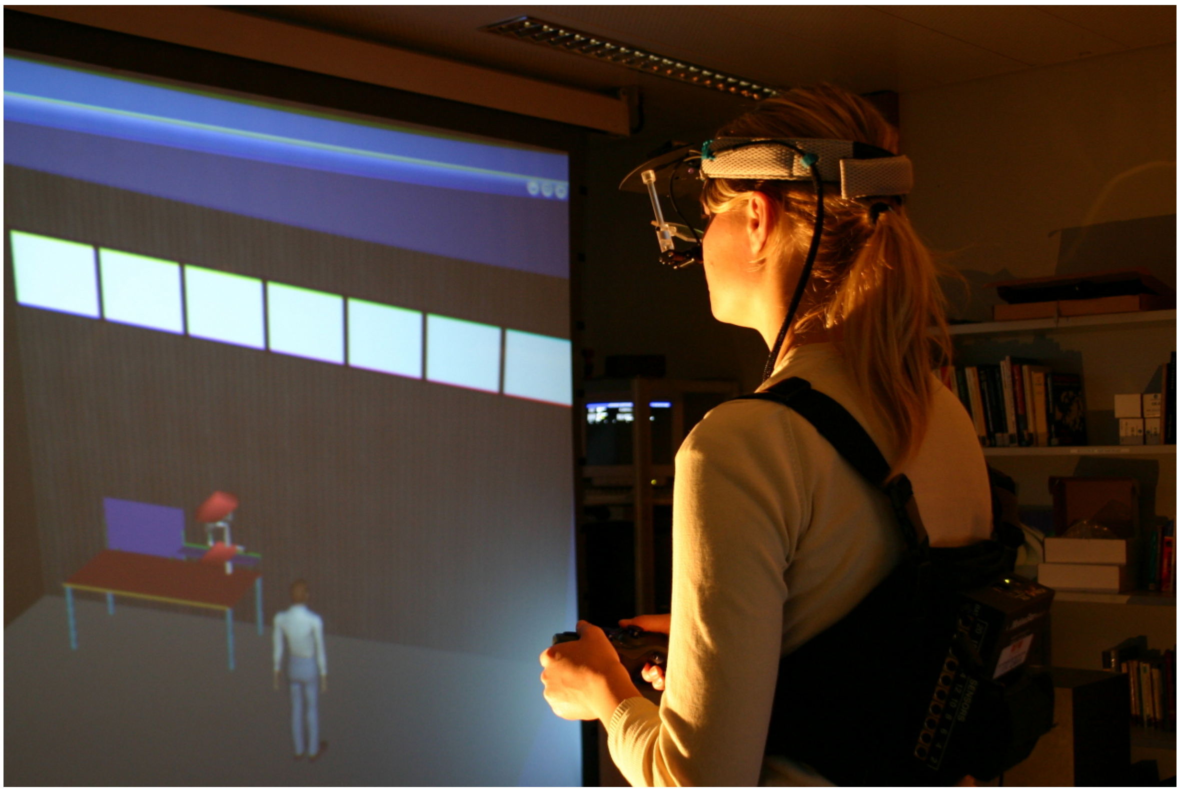


Figure 7.3: Gaze tracking experiment

The eye tracker interface was a bit of disappointment for various reasons. While it was well possible to take control of the virtual character using the ghost interface and the multi-agent framework performed well, prolonged utilization of the system proved to be very tiring for the user. Moreover, the hardware itself has frequent robustness and accuracy problems. Ideally, the system should be used only in a passive, monitoring role, not as an active input device.

7.2 Virtual guide

The virtual guide scenario puts the user in front of a virtual museum/gallery. The goal is to explore the museum with the help of a virtual guide – a robot – which is able to guide the visitor to various places of interest and provide the user with information about the works on display.

The guide is able to find the user in the museum (for example if one gets lost) and to guide the visitor to the desired destination using gestures and synthesized voice. If the user gets lost or the robot loses contact with the user's avatar then the robot returns to pick the user up again and resume guiding. The robot delivers short information about objects of interest being passed on the way to the destination and explanation about the destination itself, once there. The information is provided using synthesized voice generated using the Festival system[1].

[1]`http://www.cstr.ed.ac.uk/projects/festival/`

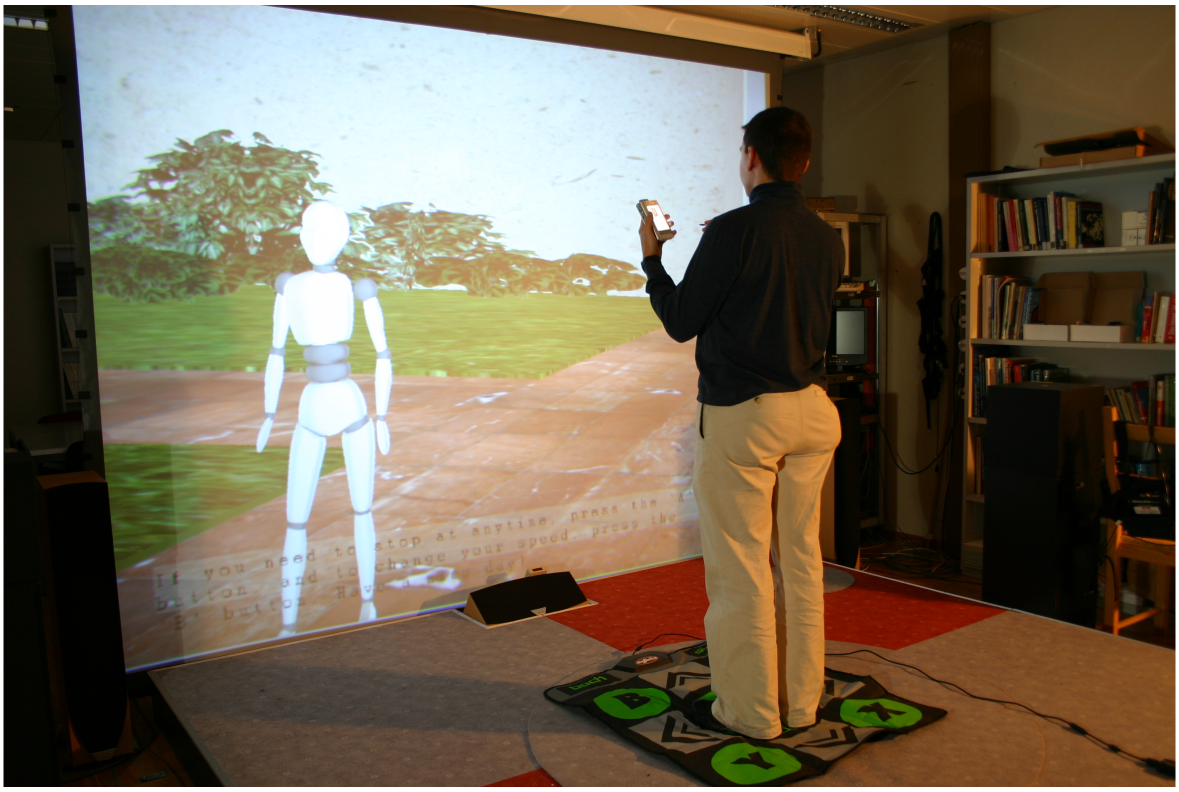


Figure 7.4: User with the virtual guide

Figure 7.4 shows the user operating the virtual guide installation. The system consists of a large back-projection screen, loudspeakers for speech output and multi-modal input system utilizing a tactile mat and a handheld computer (see fig. 7.5).

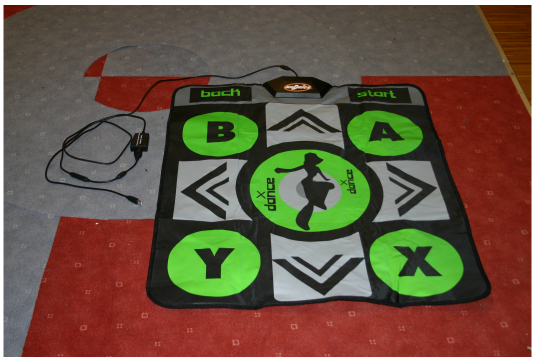
The tactile mat is used for navigation by the user – by stepping on the arrows, the user is able to move in the desired direction. In our case, the XBox$^{\text{TM}}$ mat was used with a USB converter, in order to be able to utilize it with a regular PC.

The handheld computer with a touchscreen displays a menu with pictures of the works displayed in the virtual gallery and a buttons used to instruct the guide. The handheld computer is connected by a Bluetooth link to the simulation machine in order to retain the freedom of movement of the user.

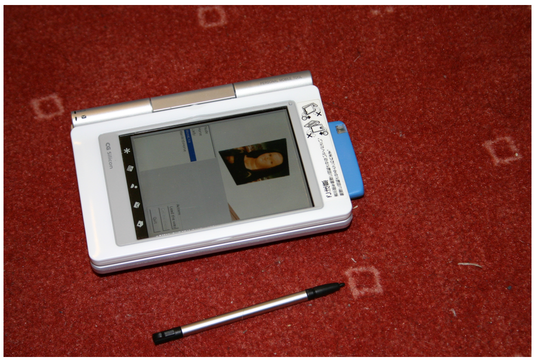
Figure 7.6 shows examples of works in the virtual museum – the pictures and the statue are reproductions of the works of the Colombian artist Fernando Botero, which were "installed" in the virtual museum.

From the implementation point of view, the system utilizes two ghost agents driving two virtual characters. One ghost–puppet pair drives the robot guide and the second one is user control ghost agent and an invisible avatar of the user carrying the camera. The rest of the framework is standard – one planning agent with the Sensory Graphplan planner, one grid agent for path planning and facilitator are used.

To localize the user the guide agent uses the information made available to the facilitator by the user's ghost agent. Then the path planning agent (a "grid" agent) is used to compute a path to the user and the guide will follow it until it arrives at predefined minimal distance from the user ($\approx 1m$).

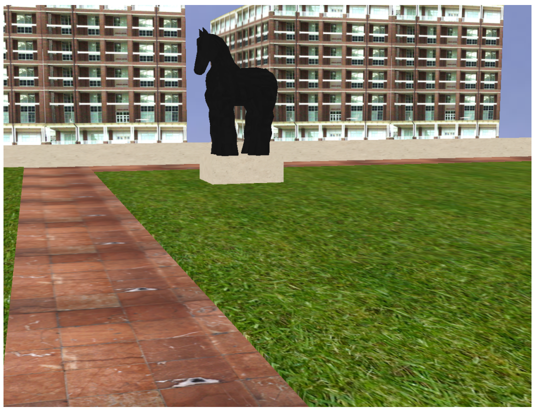


(a) Tactile mat

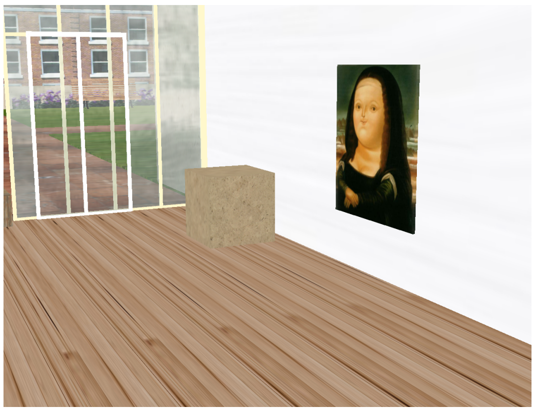


(b) Sharp Zaurus PDA with user interface

Figure 7.5: User interface for the Virtual Guide scenario



(a) Horse statue



(b) “Mona Lisa”

Figure 7.6: Virtual museum exposition

In order to be able to symbolically express the goals of the user and the orders to the guide agent, the environment had to be augmented with semantic information defining the names of the places, which places are linked together (in the navigation sense) and where are the objects of interest. Part of the semantic information for the museum is shown in figure 7.7.

The semantic information in the application consists primarily of geometrical information defining several large “rooms” which are used to describe the position of the agent in coarse terms (such as “I am at the horse” or “I am inside”) and topology information, describing the connections between the rooms. There is a missing link between the “frontyard” and “lobby” rooms, the omission is intentional, because there is a front door separating these two rooms which defaults to the closed state. Upon opening the door, the link between the rooms will be created dynamically.

The yellow points highlighted in one of the rooms define arrival/exit points – these points are used by the agent as start/end points for path planning to/from the room.

The locations of the artworks are described by named points – sets of 3D coordinates with a name associated representing points of interest. The approximate location of each artwork is specified by

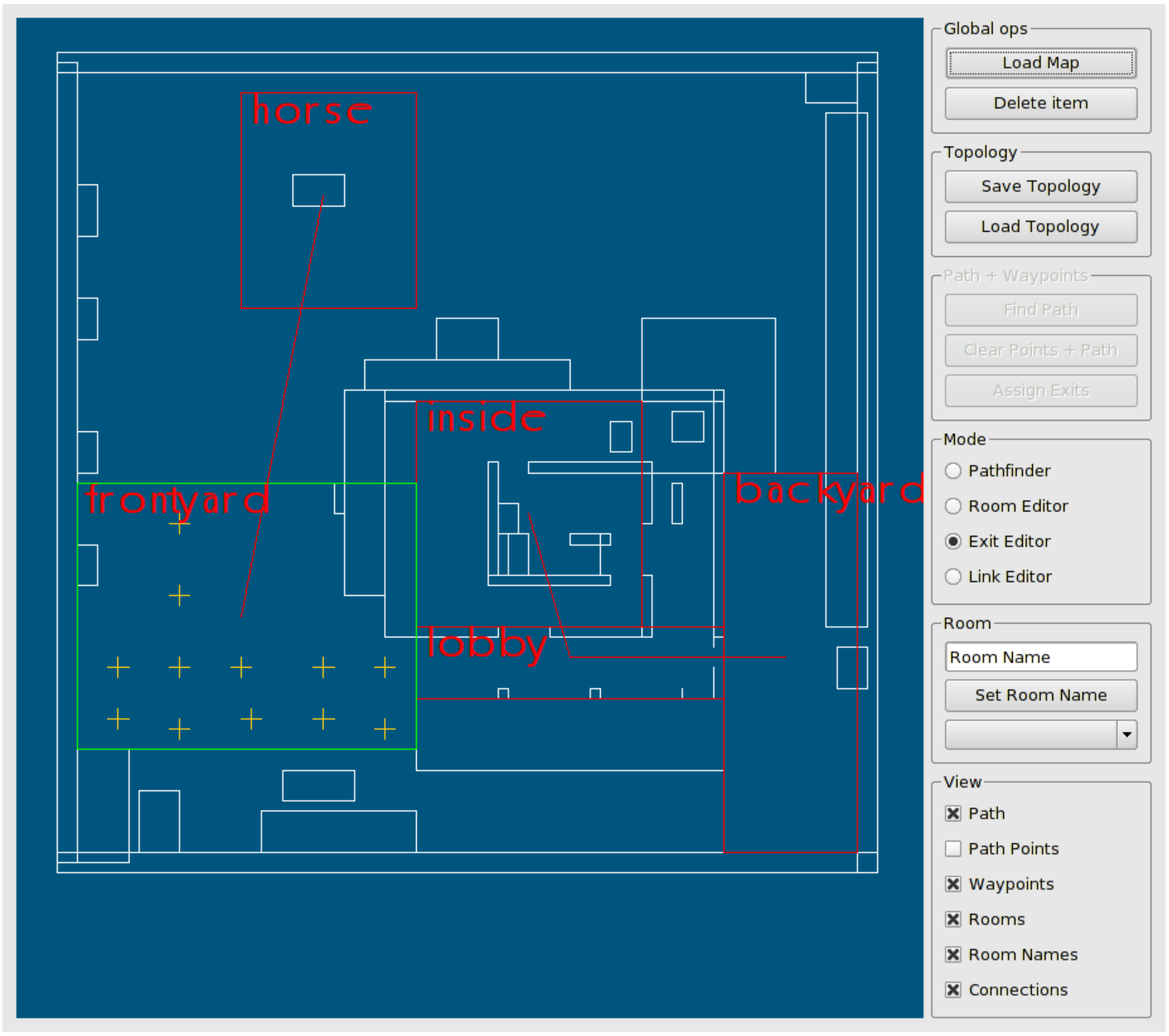


Figure 7.7: Semantic information for the museum

saying in which room the artwork is. Therefore the path planning process is actually a two-level problem – the agent has to find the way to the correct room (determine which rooms have to be traversed), this is usually done as a part of the action plan described below. The second part is to find the geometrical path between the rooms and once in the room with the goal, to find a path to approach the goal.

The reason for the two-level planning approach is that the agent is able to take into account that he has to open/close doors or to deal with the human visitor. A pure path planning approach would not be sufficient for this purpose because most path planners do not work well with a changing topology of the world (updates of the adjacency graph used for path planning due to the added/removed obstacles are very expensive operation).

7.2.1 Interaction

The user interacts with the scenario in two ways:

- Directly – the user is able to explore the environment by navigating the scene using the tactile mat.
- Hybrid mode – the user can summon the guide and ask it to see a certain work using the handheld computer (by picking the work of interest from a list of images on the display).

The hybrid mode is implemented as a team-based collaborative process, where the guide agent and the user (represented by his ghost agent) work together to find the desired art work. User forms a team recruiting the help of a guide (there could be multiple guides available) and delegates the task description specifying that he wants to be near certain artwork. The guide uses the planner as described in the previous chapters to find an action plan satisfying this request and starts to execute it. The plan usually consists of three main stages:

1. Find the user and approach him – the guide agent has to discover the location of the user in the virtual environment and has to come to pick him up.
2. Guide the user towards the requested goal – the guide is showing the way to the user, which is supposed to follow manually using the tactile mat. If the user falls too far behind or gets lost, the guide agent localizes the user again and returns back to him to pick him up again. This process is triggered by evaluating the distance between the user and the guide using a path between their respective positions – if the distance is longer than a pre-defined threshold, the guide will return to the user.
3. Approaching the goal and delivering some commentary – the guide brings the user to the objective and delivers information about it.

7.2.2 Evaluation

The virtual guide scenario performs surprisingly well, even though the interaction between the user and the virtual guide may seem on the first glance unnecessarily complex. Nevertheless, the more complex solution using task delegation and planning has a large advantage in flexibility. With minimal changes, it is possible to deploy multiple virtual guides guiding both human users or autonomous agents and the system will accommodate the change without problem.

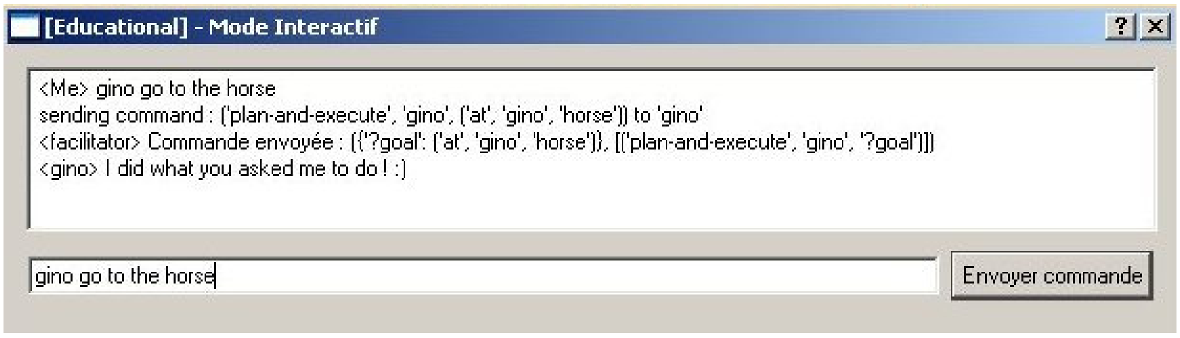
It is also possible to add or remove the points of interest, even at run time – thanks to the dynamic approach used by the guide agent, the data will be picked up and used automatically. The only required change is to update the user interface on the handheld computer with new data – for simplicity of the implementation, the data have to uploaded to the handheld machine manually at the moment.

However, while extending the scenario and adding new data, the limits of the used planner became quickly apparent. The SGP planner is unable to cope with too many connections between the rooms and too many objects in general. The final application uses five rooms visible in the figure 7.7, four connections and five artworks available for the user to visit. The guide agents uses 14 operators for navigation, guiding and to interact with environment (open/close doors, move objects). In this configuration and using compiled LISP version, the planner takes approx. 10-15 seconds to build the plan, which while still being interactive, is not realtime and pushes the limits of the planner already. The problem seems to be the very slow backward-chaining search inherited from original Graphplan and the plan forking approach to contingent planning used by SGP authors. To mitigate this wait for the human user, an additional voice announcement was added informing the user that the guide was summoned and will arrive shortly.

7.3 Natural language interface

Human users do prefer to communicate with the machine on their own terms, using familiar language. An experiment was performed to create a natural language interface to control the autonomous agents in the simulation.

The facilitator-based system permitting to delegate tasks to the agents is a very good candidate for supporting a natural language interface. The tasks are delegated in a form of logical expressions which can be mapped to the natural language. In this case, the mapping was implemented for limited form of English language. The interface is implemented as a translator agent.



```
<Me> gino go to the horse
sending command : ('plan-and-execute', 'gino', ('at', 'gino', 'horse')) to 'gino'
<facilitator> Commande envoyée : [{'?goal': ('at', 'gino', 'horse')}, [('plan-and-execute', 'gino', '?goal')]]
<gino> I did what you asked me to do ! :)
```

```
gino go to the horse
```

Figure 7.8: Translation from written English to task specification

Figure 7.8 shows an example of such translation. The user typed an order for the specific agent: “Gino, go to the horse!”. This order will be translated to `(plan-and-execute gino (at gino horse))`. It is also possible to give orders for multiple agents or for everybody at once – such as “Everybody go to the lobby!” translated into `(plan-and-execute ?a (at ?a lobby))`.

The translation process is done in several steps using word taxonomies, expression normalization and knowledge about the construction of valid English sentences, such as that each sentence has a verb, a subject performing the action and usually a complement determining the object the action is being performed on or giving additional information about the action. These sentence constructs are identified and used to build a goal specification for the agent.

7.3.1 Interaction

The user interacts with the system by typing English sentences which are then translated on the fly into the goal specifications for the system. The user interface agent communicates directly with the facilitator and delegates the built statement to the it for processing and delegation to the agents. Both facilitator and the agent(s) the task was delegated to report the task status back to user in order to indicate whether it succeeded or failed and what was the reason for the failure.

The described user interface works on a high level using direct orders to specific agents. The agent tries to build a plan to satisfy the given goal and then operates according it. Depending on the goal given and the resulting plan, it is possible that the agent will create a team to satisfy it, however this all happens without user’s intervention.

7.3.2 Evaluation

Unfortunately, the usability of the described translator is not very good, because of the amount of data that has to be prepared for every scenario – the word typing (e.g. "gino" is a "person", "go" is a verb etc.), normalization of certain expressions (such as "pick up" has to be a single keyword `pick-up` in order to match the verbs with available actions) and the sentence patterns used to generate the task specifications. These data have to be prepared for each scenario anew and care has to be take to ensure that they are consistent with the rest of the system (the action names match, the agent names match, etc.). Moreover, the algorithm used is not very robust, it manages to recognize certain types of sentences very well, but fails badly on others.

The result of this experiment shows that there is a potential for a natural language input translator, but some more work is needed. The performance and usability would improve a lot by integrating a proper natural language processing engine and ideally making it get the required information from the facilitator directly instead of having to define everything manually.

7.4 User interface for problem solving

Interaction with the complex simulation system consisting of many agents can be a daunting task without an appropriate interface. Usually these interfaces are designed in an ad hoc application/scenario specific manner, such as the specialized interface to the City Riot scenario presented in the section 7.6.

On the other hand, it is useful to have a generic interface which can be used even if there is no scenario-specific one or if it is too restrictive. The interface has to allow visualization and inspection of the current state of the simulated world, specification of the tasks to be done and at least some basic sanity checking for the tasks being specified by the user. Such tool is especially useful during the development of the scenario and in the debugging process.

This case study intends to demonstrate how a user interface can exploit various capabilities of the collaboration framework presented in this thesis to provide a meaningful but completely scenario-independent tool to the user.

7.4.1 Interface

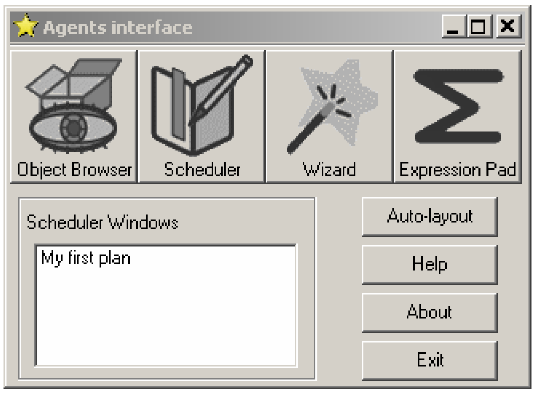
The interface presented to the user consists of several windows, as seen in figures 7.9 and 7.11. The start screen offers four main tools available to the user – the object browser, scheduler, the shortcuts window (also called the "wizard" window) and the expression editor.

The user interacts with the running simulation using either imperative orders or goal specifications. These are built using the facilities provided by the interface and dispatched to the facilitator for delegation to the agent(s) to be executed.

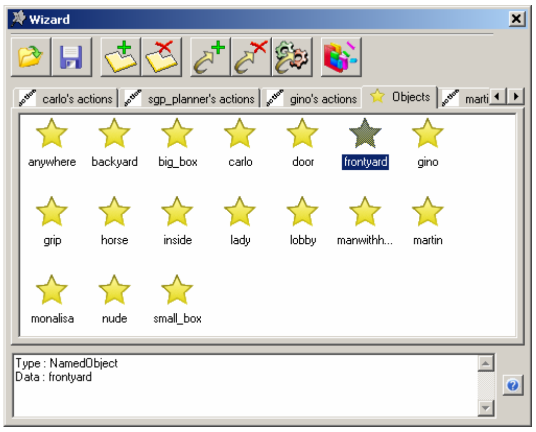
7.4.2 Introspection

The ability to inspect the state of a running system is the key to any meaningful interaction in a collaborative environment. Without this ability the user would be unable to know what are the agents capable of and what is their current state such as whether they are free/busy or their location.

This functionality is implemented using introspection – the ability to examine running objects in the Python language. Python has excellent introspection capabilities, however in this case they need to be extended in two ways:

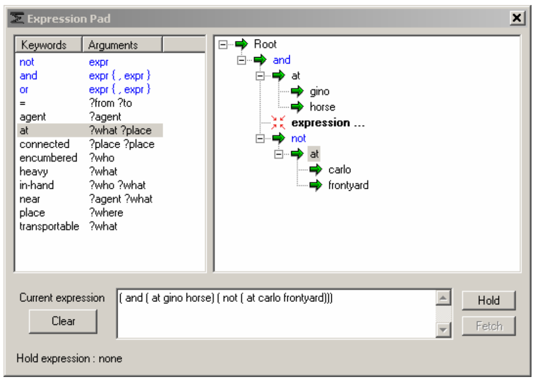


(a) Start screen



Type : NamedObject
Data : frontyard

(b) Shortcuts screen

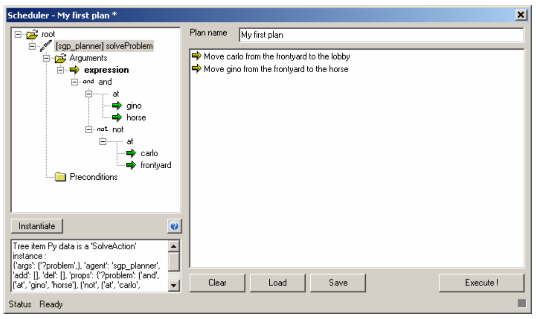


Keywords	Arguments
not	expr
and	expr { , expr }
or	expr { , expr }
=	?from ?to
agent	?agent
at	?what ?place
connected	?place ?place
encumbered	?who
heavy	?what
in-hand	?who ?what
near	?agent ?what
place	?where
transportable	?what

Current expression (and (at gino horse) (not (at carlo frontyard)))

Hold expression : none

(c) Expression editor



Plan name My first plan

Move carlo from the frontyard to the lobby
Move gino from the frontyard to the horse

```
Tree item Py data is a 'SolveAction'
instance :
{'args': ('?problem',), 'agent': 'sgp_planner',
'add': [], 'del': [], 'props': {'?problem': ('and',
('at', 'gino', 'horse'), ('not', ('at', 'carlo',
```

Status Ready

(d) Scheduler

Figure 7.9: User interface for problem solving

- The state of the agent and its capabilities need to be examined. Python introspection works on lower level, for example the names of the functions can be retrieved or currently defined variables can be examined. Therefore, the required higher-level functionality (such as enumeration of agent's capabilities and beliefs) has to be built in terms of the Python capabilities first.

- The introspection functionality needs to be accessible from outside of the agent's process. Typically, the system is distributed and the information has to be accessible remotely. For this purpose a special CORBA interface was implemented.

The required introspection capabilities were built using the information available to the agent – its state and the information gained from the functors implementing the actions.

Figure 7.10 depicts how the introspection layer is implemented as an additional CORBA interface extending the regular FacilitableAgent. The methods of this interface provide direct access to the lists of actions (functors) with all their metadata encoded using the Python "pickle" mechanism to marshal them for easy transport over CORBA. The same approach is used for the agent's beliefs. The `checkSolvable` function is used to verify an individual predicate, whether it holds or not in the context of the agent's beliefs.

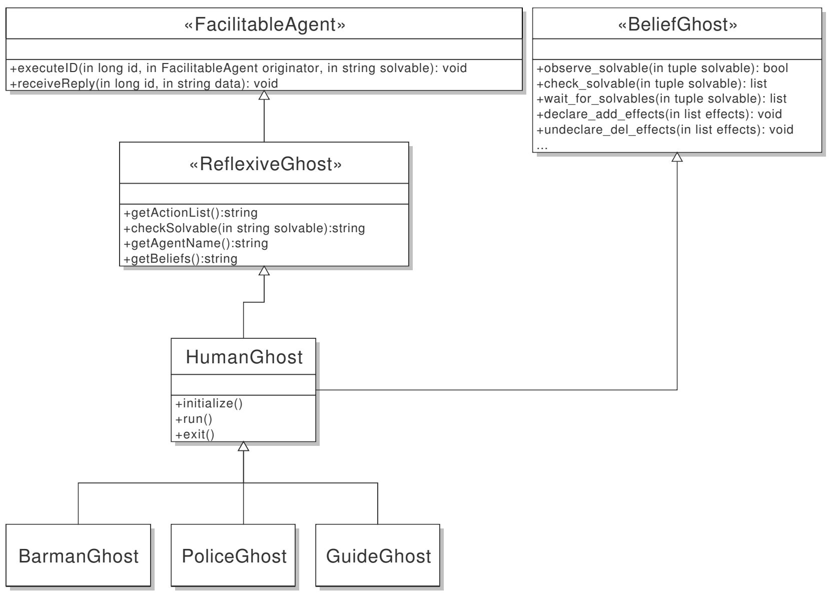

«FacilitableAgent»
+executeID(in long id, in FacilitableAgent originator, in string solvable): void
+receiveReply(in long id, in string data): void
«BeliefGhost»
+observe_solvable(in tuple solvable): bool
+check_solvable(in tuple solvable): list
+wait_for_solvables(in tuple solvable): list
+declare_add_effects(in list effects): void
+undeclare_del_effects(in list effects): void
...
«ReflexiveGhost»
+getActionList():string
+checkSolvable(in string solvable):string
+getAgentName():string
+getBeliefs():string
HumanGhost
+initialize()
+run()
+exit()
BarmanGhost
PoliceGhost
GuideGhost

Figure 7.10: Autonomous agent with introspection class diagram

Similar introspection is also provided for other agents (software agents in general, not just ghosts) available via CORBA in the current simulation system. The agents are enumerated together with their methods and displayed in the form of a list – see figure 7.11. For agents with static interfaces, the methods can be retrieved directly using Python, unlike for the ghosts, where the interface can change depending on whether some new object-specific actions were loaded or not or which default actions were loaded at the start of the agent.

The object browser provides the user with multiple information:

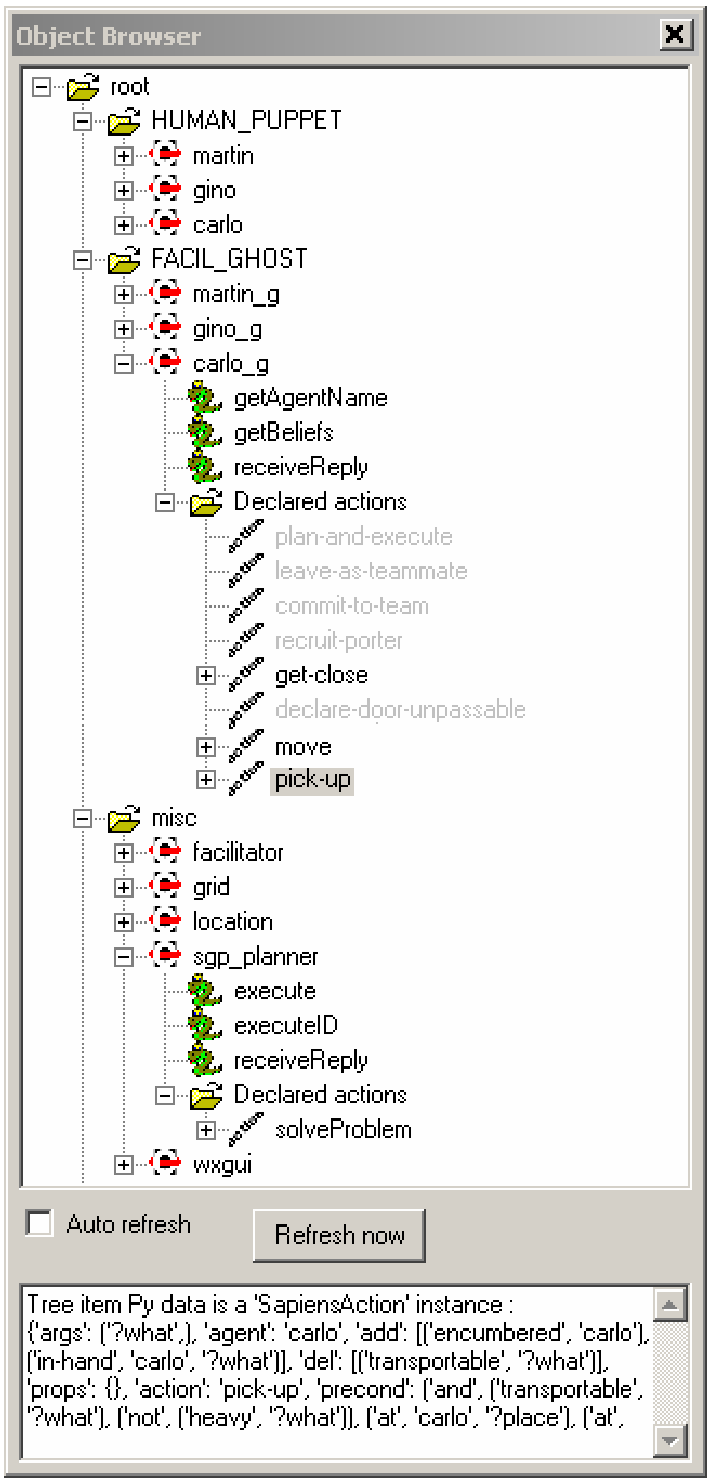
- List of running agents and their types (ghost, puppet, something else).
- For each non-ghost agent its static API and its facilitator-declared capabilities are displayed.
- For each ghost agent, there is the static API and its facilitator-declared capabilities displayed. The facilitator-declared actions are checked against the agent's current state and impossible actions (with unsatisfied preconditions) are grayed out.

The information in the object browser is updated to reflect the current state of the system and provides the user with a quick overview of the situation.

7.4.3 Action scheduling

In order to have an agent execute an order or to complete a complex task, the goal has to specified. Using the described interface, it is possible to instantiate an action (perform a valid substitution of the arguments – the user pick out of the list of instances which are possible at the moment for the agent) and schedule this action for direct execution in the scheduler window. In this way it is possible for the user to create a *plan* for the agent(s) manually and then have it executed.

Another, more flexible, option is depicted in the figure 7.9(d). The plan is generated automatically using the action planner from the current world state retrieved from both the facilitator and the agents



```
Tree item Py data is a 'SapiensAction' instance :
{'args': ('?what',), 'agent': 'carlo', 'add': [('encumbered', 'carlo'),
('in-hand', 'carlo', '?what')], 'del': [('transportable', '?what')],
'props': {}, 'action': 'pick-up', 'precond': ('and', ('transportable',
'?what'), ('not', ('heavy', '?what')), ('at', 'carlo', '?place'), ('at',
```

Figure 7.11: Object browser

and the specified goal. This is done by direct invocation of the `solveProblem` method of the planning agent, available from the object browser.

The plan, created either manually or automatically, is displayed in the scheduler window and available for the user to modify it or to start executing it. During the execution of the plan the application assumes the role of the coordinator, delegating the individual tasks to the facilitator for dispatching to the agents for execution and tracking its progress by evaluating the incoming status reports from the agents.

Another possible option is to store the built plan as a "shortcut", as seen in figure 7.9(b). The shortcuts can be invoked by a simple click of the mouse and are typically used for frequently used actions (essentially a macro).

Another use for the shortcuts is to provide a simplified interface for non-expert user – the user will be able to use only the predefined set of actions presented on the screen in the form of shortcuts created by the scenario designer.

7.4.4 Action pre-validation

In order to ensure that the user is able to submit only valid orders or tasks to the agents, it is necessary to pre-validate his input. Of course, the agents test the preconditions of each action before attempting to execute it and the consistency of the state of the virtual world will be maintained even with invalid input. However, the user will have to wait potentially long time while the action/plan is being executed, only to learn that a mistake was made and some action is not possible in the given state, indicated by the received failure status report.

The action pre-validation is performed in two contexts:

- On action instantiation (static check)
- Before plan execution (dynamic simulation)

Static pre-validation on action instantiation uses the list of all possible argument substitutions for the given action to filter out impossible actions. For each instance the preconditions are evaluated using the current system state as retrieved from the facilitator and the impossible instances are removed. If there is at least one instance left, the action is deemed possible, otherwise the action is grayed out in the browser.

The same action pre-validation process is used when an action is being instantiated manually for insertion into the plan. In that case the user is only given list of instances which are possible (preconditions are satisfied) in the current state, making it impossible to insert an invalid action into the plan.

The pre-validation process has $O(m \times n)$ complexity, where m is the number of actions to be tested and n is the number of facts received from the facilitator. This process has the potential to be slow (essentially quadratic complexity) and more optimization is needed, such as memoization of already tested and known to be true predicates during processing of the actions and more efficient storage of the data to speed up the processing.

In case that only automated plan generation was used, no more plan validation would be necessary – the planner guaranties that the generated plan is correct with respect to the given initial and goal state. If manual plan creation/modification is allowed, this guarantee is not there anymore because the static action pre-validation described above cannot check for dynamic effects introduced by the actions before the tested action in the plan – the static check tests only again the current system state.

To address this problem a plan execution simulator was implemented. The simulator receives a copy of the current world state and simulates execution of the plan by sequential evaluation of the preconditions of the actions and application of their effects to the simulated state. Using this approach it is possible to identify problems in the plan before submitting it for execution. The invalid action will fail during the simulated execution step because its preconditions will not be satisfied.

Dynamic and static action pre-validation is able to catch the most common problems, however it has to be taken into account that the plan may still fail during the execution – for example because of the unknown side effects of the delegated actions. The pre-validation provides support to the user, but it is not a solution for every possible problem.

7.4.5 Task delegation

The plan execution is implemented using the task delegation via the facilitator. The actions are delegated to the facilitator in sequence and the facilitator dispatches them for execution, using the standard delegation approach described in previous chapters.

There is one problem which is unique to the user's view of the world state while using the described interface. For example, in the "Virtual guide" application, the user is collaborating with the autonomous agent on a peer-to-peer basis, having only partial view of the world state (partial observability). On the other hand, the described interface works directly with the global state retrieved from both the agents via introspection and from the facilitator by direct queries. A problem occurs when the plan is created relying on some information which is available to the user from the global state but is not available to the agent executing the action.

Let's consider an example in the museum scenario from the "box world" experiments:

1. Agent Carlo opened the museum door (for example as a result of a previous plan or a side effect).
2. The user specifies that agent Gino should close the door of the museum. From the user's (and the pre-validation) point of view, the action is possible, because the facilitator *knows* about the change made by Carlo (door is open).
3. The door closing action is delegated to agent Gino via the facilitator and fails.

The action/plan failure in this case stems from the difference in the world state view available to the user through the interface. The user has an unrestricted access to the world state in the facilitator (full observability), whereas the agents have only partial observability. The nature of the issue is similar to the problems caused by the side effects of the delegated actions.

There are two possible solutions for the problem:

- Tell the agent the missing information.
- Ask the agent to sense for it.

The second solution is preferable and was actually implemented, because the sensing is also useful to mitigate the problems caused by the side effects. Another reason is a problem of accidental "poisoning" of the beliefs of the agent, if fed with incorrect information.

Sensing was described in the section 6.3.4. However, by default the agent senses only the content of the current room (list of objects contained there) and tests the solvables explicitly contained in the sensing list, which may or may not contain the required solvable (e.g. the door state from the example above), causing the inconsistency.

In our case, the agent modifying the state of the world has the option to add the modification to the list of solvables to test for in the current room. Alternatively, the solvable can be put there by the scenario designer or the solvable can be sensed for every time (useful for global facts, such as time of the day). During the next observation cycle, the agents will test for this solvable and learn about the new modification – limiting the impact of the problem described above.

7.4.6 Evaluation

The described user interface proved to be a valuable tool while debugging the problems in the scenarios being developed. Especially the introspection features are very useful while debugging a fully distributed system running many agents.

The interface itself is a prototype developed more as a feasibility study than a finished solution usable by non-expert users. It would have to be simplified further to shield the user from issues such as action instantiation and rather complex plan building. The shortcut feature provides a good starting point to create specialized, high-level interfaces to the concrete scenarios, without complex programming skills required.

From the interaction point of view, the interface provides the user with essentially complete freedom. The user can control directly the puppets with the interface acting as a ghost agent, even though doing so is not very practical from the usability point of view – the actions need to be instantiated with arguments and for puppets there is little introspection and validation support.

The other two options were described already – the user can interact with the scenario using either manually or automatically built plans. The plans may include team forming by the agents. The user usually does not have to form a team explicitly to achieve his goal, even though the option to do so is there, albeit laborious – the user would have to form a team manually by scheduling the right actions and than manage the team as well. It is a lot easier to just delegate the full task to some other agent which will perform the tricky job instead.

Overall, the application is a good demonstration of possibilities available to the developer using the collaborative agents framework. The introspection, action validation, plan creation and the shortcut macros are techniques which are valuable for other scenarios as well.

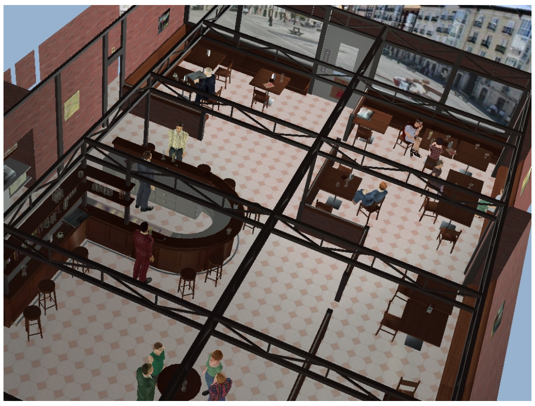
7.5 Declarative "story specification" for VR exposure therapy

In the field of psychotherapy, cognitive and behavioral therapists started to use the virtual reality for the exposure sessions to complement a therapy *in vivo* for many psychological conditions. It has been widely applied to various phobias, such as fear of flying, arachnophobia and acrophobia. More recently, it has been used to treat also the social anxiety disorder – the patient is unable to function normally in social situation because it makes him anxious (such as speaking in front of an audience, meeting friends in a bar, etc.).

The challenge for the therapist is to reproduce an anxiety stimulus in the patient. In order for the virtual reality to be effective for this purpose, it has to be:

- Believable – in order to not break the immersion and to work as an anxiety stimulus in the first place.
- Reproducible – the scenario has to be fully repeatable, essentially ruling out fully autonomous systems. It is not sufficient that the agent will perform the same activities if they do not happen

(a) The bar scenario


(b) The bar scenario

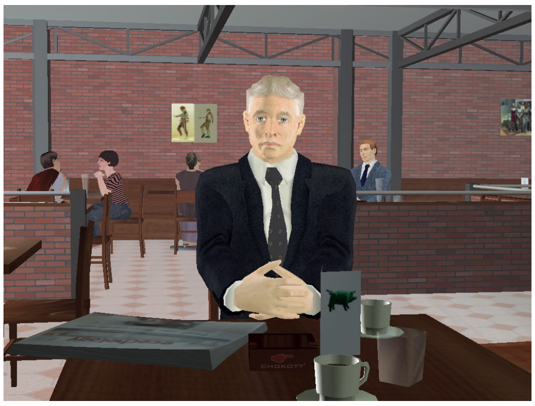

(c) Patient's view

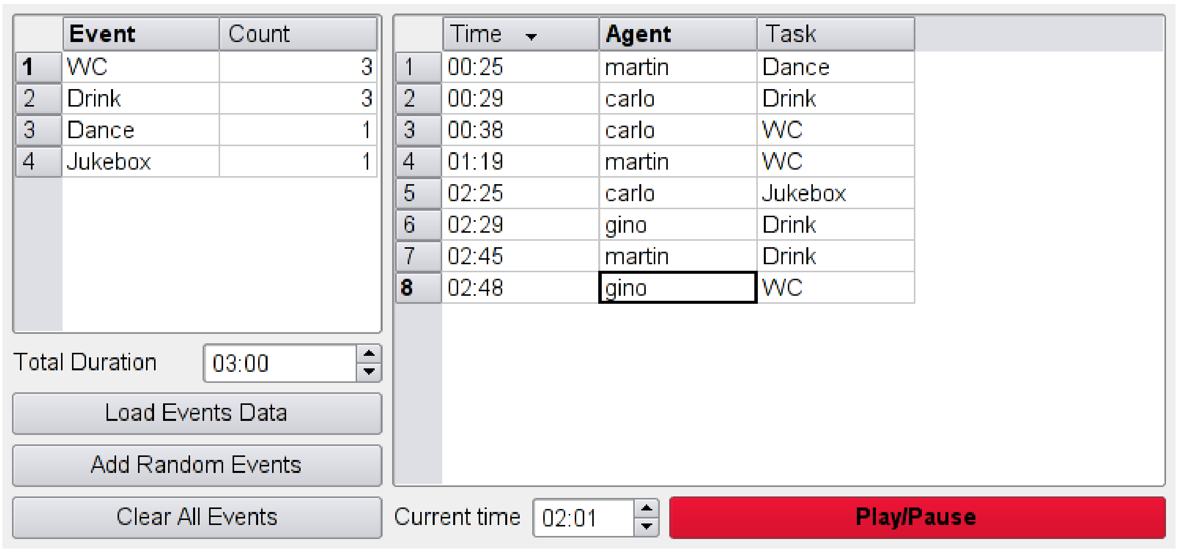
Figure 7.12: The bar scenario

(for example) in the same place and in the same situation. If the scenario is not fully repeatable, it is impossible for the therapist to evaluate/compare the results obtained.

- Controllable – the therapist has to be able to intervene at any moment and be able to adapt the scenario to the condition of the patient.

In the typical case, the therapist is a non-expert user and the application has to be as robust and simple to use as possible. A typical VR system is usually built from the machine perspective – the interaction is low level in the style: "in order to achieve this, you have to do this, this and that'. Far easier for the non-expert is to use a system which offers human perspective: "I want this this and that to happen, arrange it!" The actions should simply happen, without the need for complex scripting or programming.

The collaborative framework described in this thesis is a good candidate for implementation of such simulation system because it offers high level actions much closer to the interactions with the real world – such as "go to the toilet" implying that the agent will negotiate doors, flush the toilet after use and return to the original place. The reproducibility and controllability are also easy to achieve by using the ghosts & puppets framework.



	Event	Count
1	WC	3
2	Drink	3
3	Dance	1
4	Jukebox	1

Total Duration 03:00

	Time	Agent	Task
1	00:25	martin	Dance
2	00:29	carlo	Drink
3	00:38	carlo	WC
4	01:19	martin	WC
5	02:25	carlo	Jukebox
6	02:29	gino	Drink
7	02:45	martin	Drink
8	02:48	gino	WC

Current time 02:01

Figure 7.13: Action scheduler

To test the possibility of using the described framework in a therapeutic setting, a special scenario for treatment of social phobia was created. The scenario takes place in a bar full of people (figures 7.12(a), 7.12(b)). The patient is supposed to meet a friend there and make small talk with him/her (figure 7.12(c)).

In order for the scenario to be believable and have the desired effect, the virtual characters have to "be alive" – the waiter should arrive and offer drinks, some people may dance to the music and some simply go to the toilet and back.

The amount of this "disturbance" present has a direct effect on the patient's anxiety and it is therefore desirable to have a simple means to control it – for example the patient starts in a very calm bar where everybody is drinking coffee and the patient is undisturbed. As the therapy progresses, the patient may be gradually exposed to more stressful situations – the interaction with the waiter, people playing music on a jukebox etc.

One possibility how to regulate the amount of "action" in the scenario is to script everything. This is commonly done and allows to have a total control over every aspect of the simulation, however it is extremely labor intensive and unfeasible for the therapist himself.

A better solution is to use high level actions to tell the virtual characters what to do and let them work out the details autonomously. The therapist only has to specify the amount of the actions to be performed, for example: during the session, the waiter will inquire about order twice, somebody goes to the toilet and some people will dance.

Using the capabilities of the collaborative agent framework, implementation of such automated background "story" specification is simple. Figure 7.13 depicts a simple action scheduler, where in the left part of the window, the therapist can specify how many times each action should happen and the total duration of the scenario. Afterwards, a random schedule conforming to the given specification will be generated and another push of a button starts execution of the generated scenario.

The events available for the therapist are defined as declarative task specification, for example the "Drink" event is defined as task: `(plan-and-execute %a (not (thirsty %a)))`. This is in fact a *template*, the `%a` will be substituted for a randomly chosen agent. In case, that the some

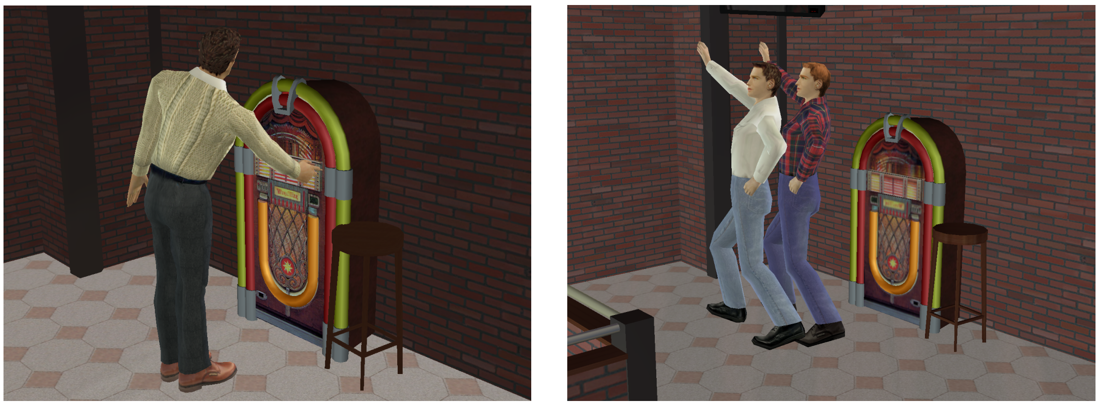

(a) Jukebox (b) Virtual characters dancing

Figure 7.14: The "dance" action

action (event) should be always performed by a certain (not random) agent, the event can be defined with the name of the agent explicitly specified. During the execution of such generated scenario, the tasks are sequentially delegated to the corresponding agents using the already described procedure.

Using the declarative task specification for the event definition has another advantage: the agents performing the tasks will use the planner. The planner ensures certain consistency, such as that a virtual character told to drink something will attempt to order a drink from the waiter, except in the case that he has a non-empty drink in front of himself already. All the details of ordering a drink (such as summoning a waiter, making an order, having the waiter bring the drink etc.) will be planned automatically by the agent and the therapist does not have to care about them. This property is in stark contrast with the scripting approach, where all this information has to be manually programmed into the scripts.

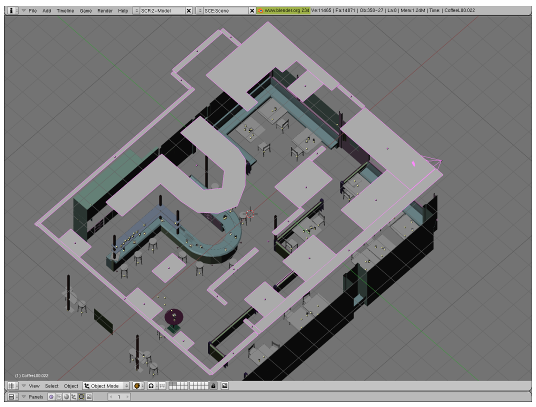
Figure 7.14 shows the "dance" action (event). Virtual character Martin turns on a jukebox and Gino together with Carlo are dancing. Apart from the use of a planner, this action demonstrates the use of object-specific actions (jukebox operation) and sensing (Marting turns on the jukebox and Gino with Carlo detect the music playing/stopped).

In order to be able to specify the required semantic information for the scenario, a small tool was created using the free 3D modeling application Blender[2]. With the use of few custom export scripts, it is possible to use Blender to define:

- Obstacles for the path planning
- Rooms with their names
- Exit/Entrance points for the rooms
- Named points

Unfortunately, it is not yet possible to define the connections between the rooms using Blender because it does not allow arbitrary links between objects. The connections/links have to be added manually afterward.

[2]`http://www.blender.org/`


(a) Obstacle definition relative to the geometry of the bar

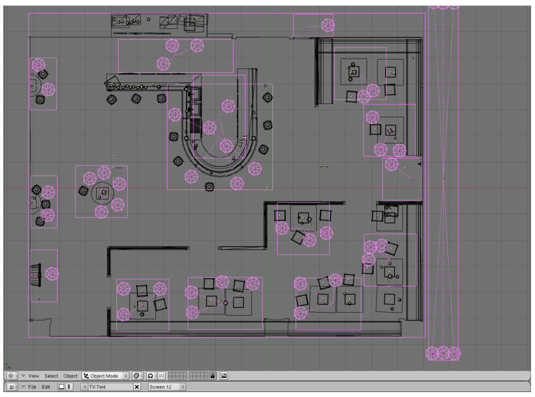

(b) Rooms with their entrance/exit points

Figure 7.15: Definition of semantic information in Blender

7.5.1 Evaluation

The automatic generation of "life" using the collaboration framework together with the action planning provides a unique way of how to automatically create new background activity, while retaining consistency and being reproducible and controllable at the same time.

The user interacts with the simulation mainly indirectly, by building a global plan of high level, declaratively defined, tasks for the agents to execute. The tasks may require further planning and processing on the part of the agents, however this is transparent to the user. No teams are explicitly created by the user, but the agents may form teams during the course of the execution of the delegated tasks.

Overall, the case study verified that the approach taken is workable in practice. It is possible to have a very high level control over a scenario (the "disturbance" level) and leave the low level details to the autonomous agents to work out.

The described scenario is not yet complete (some object-specific animations are still missing) and the field testing with a therapist and patients is planned for near future.

7.6 City riot

Various training systems have frequently specific requirements not satisfied by regular agent-based simulation frameworks. In particular, the issues of controllability and reproducibility are problematic in systems with fully autonomous agents. On the other hand, scripting-based systems suffer from lack of flexibility and difficult modification.

The presented study strives to demonstrate how a very flexible simulation system can be built using the collaborative agent framework developed during the course of this thesis. The task delegation framework provides the user with high level control of the situation and the lower level capabilities of the ghosts & puppet system allow the user to become one of the actors in the simulation.

On the high level order-based interaction the user lets the agents decide the low level details of the actions themselves. On the other hand, the user is able to perform actions as one of the agents directly on the scene, enabling him to compensate for the possible shortcomings of the artificial intelligence or to perform actions the AI is unable to do.

To demonstrate the capabilities of the developed collaboration framework, a test case simulating a training system for police was implemented. The user is put in a position of a police commander having to manage a large crowd of people moving in urban environment (fig. 7.16). The goal of the user is to prevent access of the crowd to certain places of strategic importance in order to prevent damage (such as looting or arson).

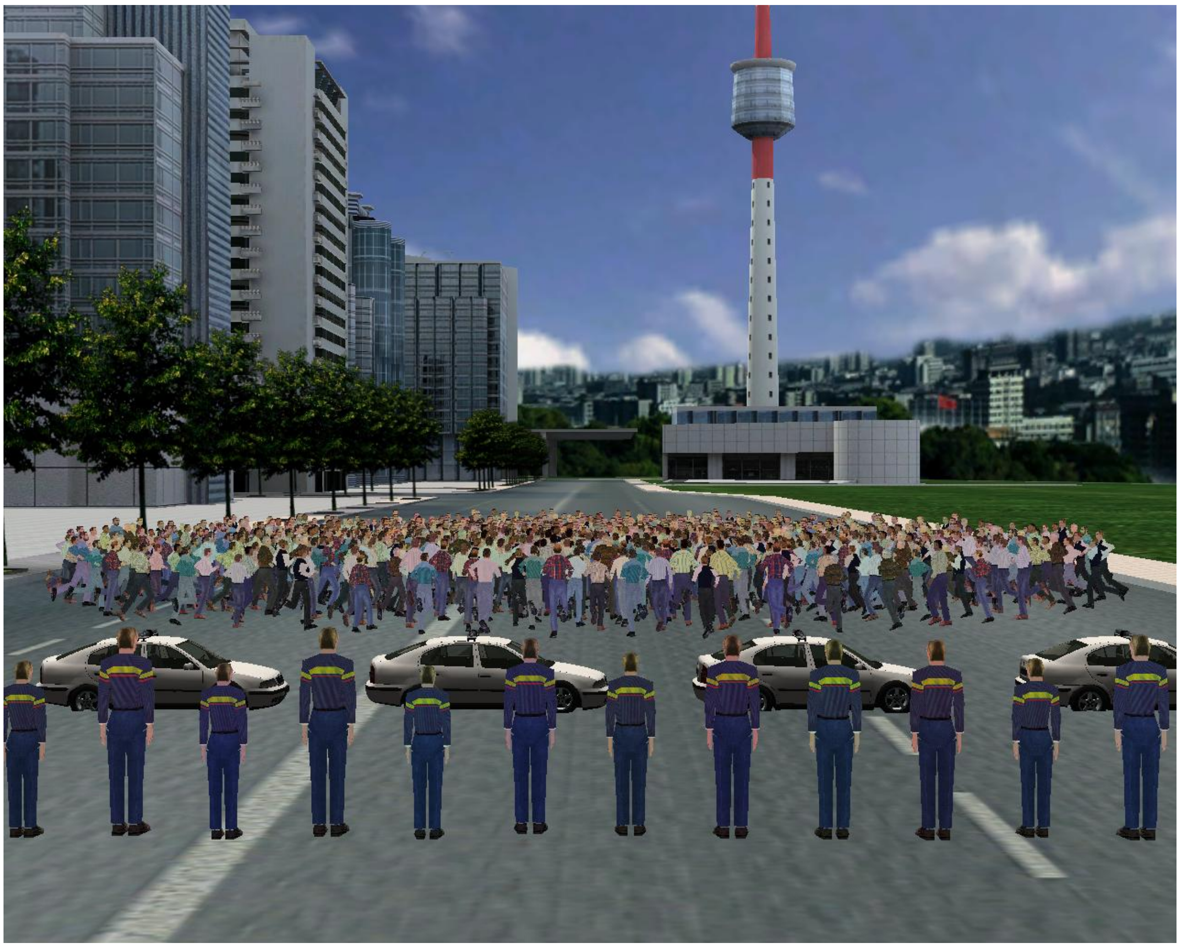


Figure 7.16: Police blocking a street from rioting crowd

7.6.1 Architecture

The application utilizes the standard architecture centered around the facilitator and the ghost agents. The planning agent is used to perform the action planning duties utilizing the fast Metric-FF planner described before. The visualization part is implemented using the freely available Delta3D[3] graphic engine.

[3]`http://www.delta3d.org/`

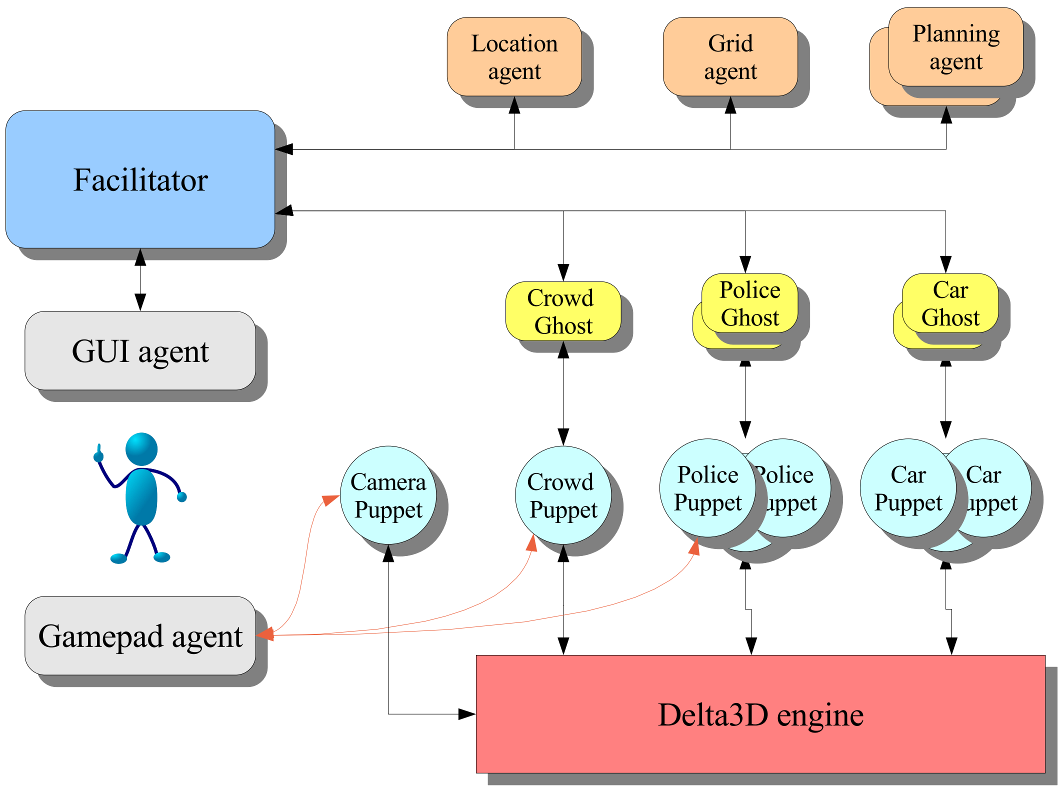


Figure 7.17: Architecture of the scenario

Apart of the standard components of the system, there are two user interface agents available – a gamepad agent for the first person interaction and a graphical user interface (GUI) agent for indirect, command-based interaction. The architecture of the scenario is shown in the figure 7.17.

There are three types of autonomous ghost agents present in the scenario:

- Crowd controlling agent. By default, the ghost only moves the crowd to a location randomly picked out of a list of several predefined places. The crowd consists of a leader and 1000 virtual characters following the leader.
- Police ghost agent. The police ghosts control the individual policemen. There are 20 individual policemen in the scenario.
- Car ghost agent. There are 20 cars controlled by separate ghost agents. The cars are used to build road blocks and to transport the policemen to destinations farther away.

Each of these ghost agents has a corresponding puppet being controlled by it. A single exception is a camera puppet, which does not have its own ghost agent and instead is controlled by the gamepad agent whenever the user takes control of the crowd or one of the policemen. The camera puppet is used to keep track about which individual policeman or the crowd is controlled by the user and to manipulate the 3D cursor (visible as a red cube in the pictures) used to take control of the individual agent.

7.6.2 Interaction

The gamepad is used in the first-person mode, where the user is able to be one of the policemen on the scene or, alternatively, controls the crowd by the means of its leader. The first-person mode is implemented as a control exchange between the ghost agent controlling the policeman/crowd and the gamepad agent, as described in section 6.1.2. The user is able to select a desired virtual character to take control of by moving a 3D cursor and then with a button push take him over. The camera changes automatically to the first person view and the user sees exactly what the virtual character sees – the user *becomes* the virtual character (policeman, crowd leader), as depicted in the figure 7.18.

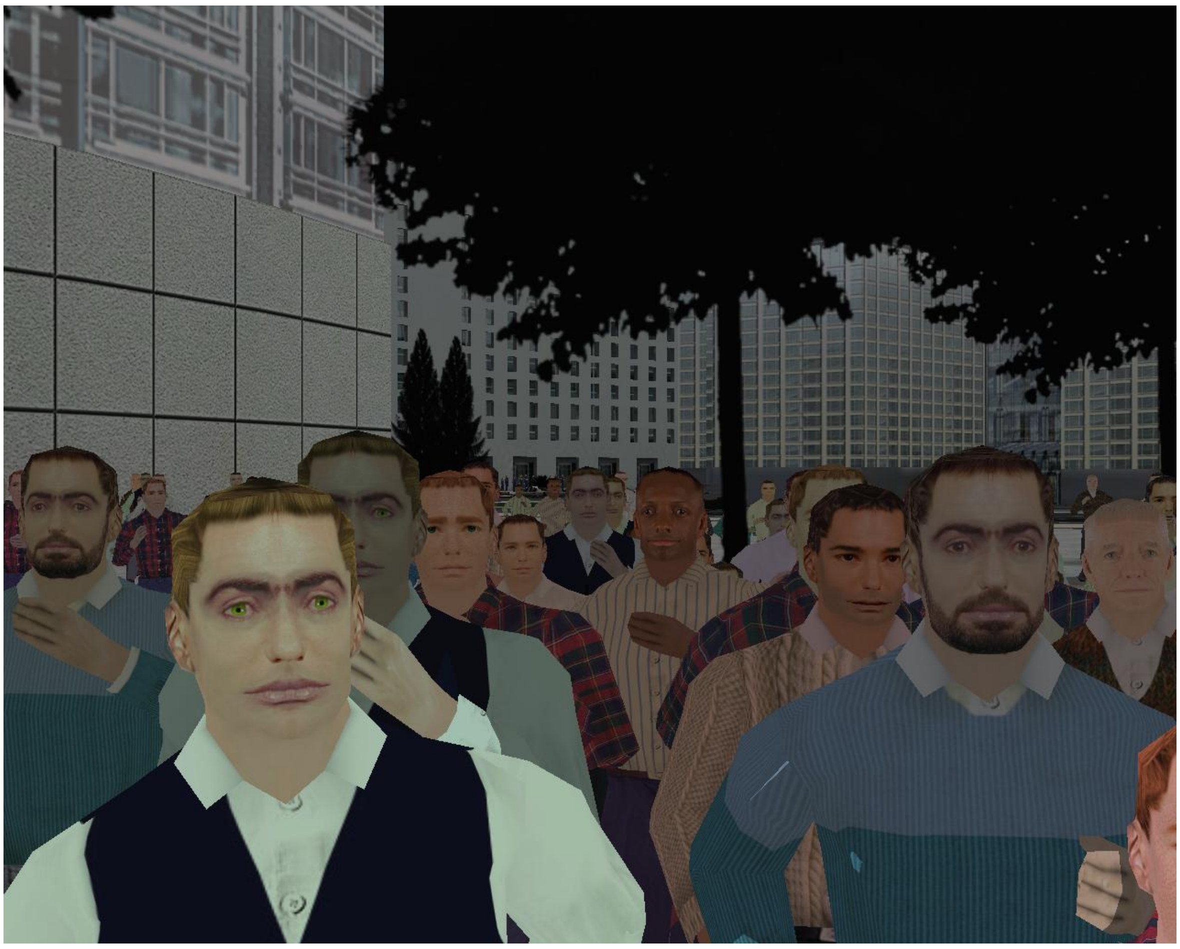
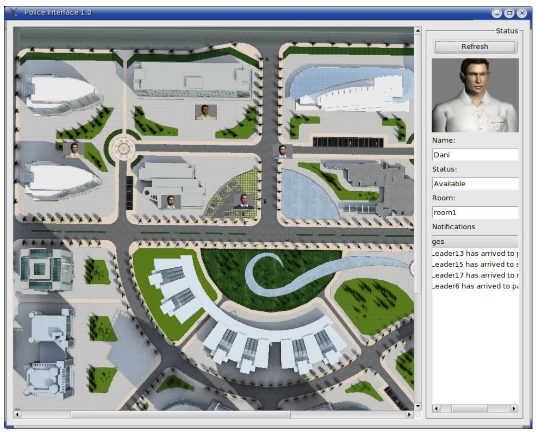


Figure 7.18: User as the crowd leader

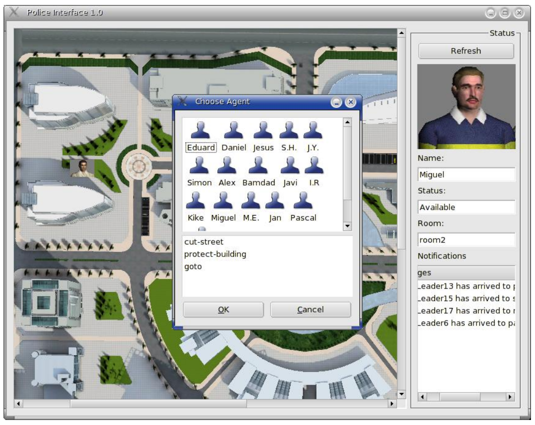
Once in the first person mode, the user is able to move the virtual character/crowd and explore the virtual environment. Alternatively, this mode can be used by the trainer to control the crowd manually, forcing the trainee controlling the police force to respond to a more challenging goal than the scripted behavior of the default crowd ghost.

Apart from the low-level "hands-on" interaction enabled by the gamepad, the trainee can use indirect, order-based control using a specialized graphical user interface agent. The graphical interface is depicted in the figure 7.19. It provides multiple functionalities:



Name:
Dani
Status:
Available
Room:
room1
Notifications
ges
.eader13 has arrived to ¡
.eader15 has arrived to s
.eader17 has arrived to r
.eader6 has arrived to p

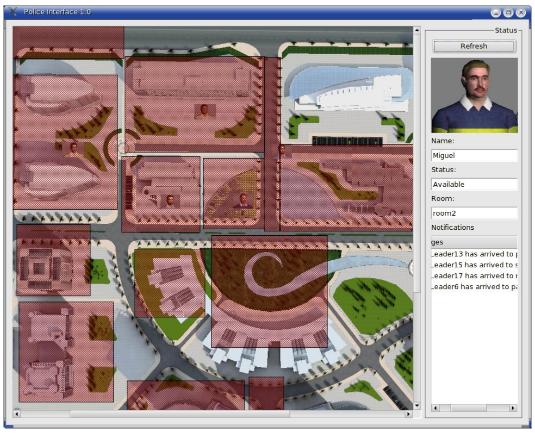
(a) Situation overview



cut-street
protect-building
goto

Name:
Miguel
Status:
Available
Room:
room2
Notifications
ges
.eader13 has arrived to ¡
.eader15 has arrived to s
.eader17 has arrived to r
.eader6 has arrived to p

(b) Dispatching orders



Name:
Miguel
Status:
Available
Room:
room2
Notifications
ges
.eader13 has arrived to ¡
.eader15 has arrived to s
.eader17 has arrived to r
.eader6 has arrived to p

(c) Selection of a place to be protected

Figure 7.19: Graphical user interface for high level control

- Agent position and activity tracking. The user is made aware of the current whereabouts of each controllable agent and its current activity by icons on the map of the city (fig. 7.19(a) and the status reports arriving to the notification area of the GUI (lower right corner of the figure).
- Order dispatch. The user is able to select an agent by clicking on the icon of the agent and assign him a task to perform, figure 7.19(b).
- Order parameters specification. Some orders (such as “cut-street”) require a place specification. The user interface enables a simple selection of the relevant location by direct clicking on the displayed map.

All orders given by the user to the agents are high level delegated actions. The agents are responsible to work out a plan leading to the successful completion of the assigned task, frequently necessitating creation of teams for this purpose.

There are two typical activities where teams are formed:

- Blocking off a location. In this case, the crowd has to be prevented from accessing given location. The blocking is achieved by creating a fenced obstacle and posting police cars and policemen on the scene.

- Moving policemen to a distant location. If a policeman has to move to a distant location (defined in terms of connections between the "rooms" in the virtual environment), he will summon a car to drive to the destination instead of walking.

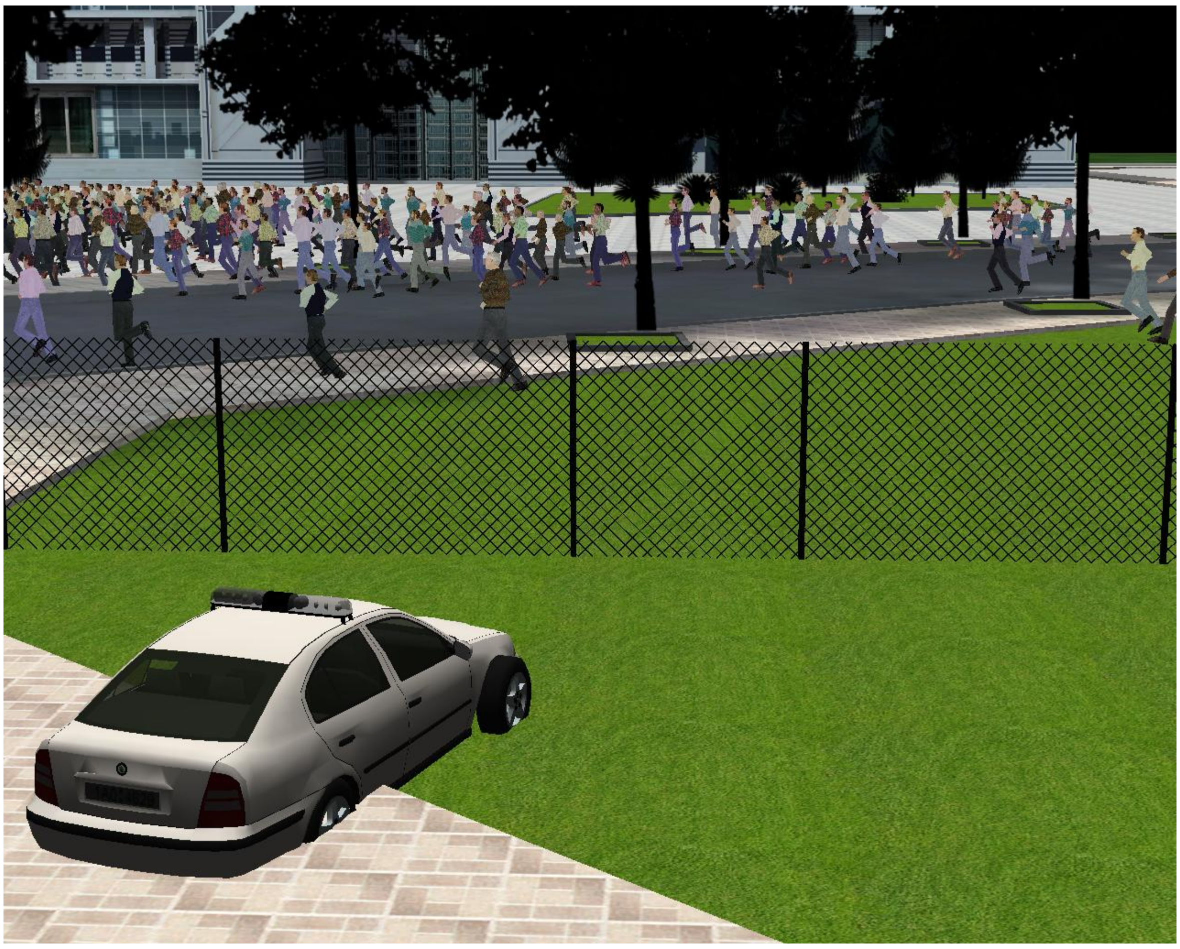


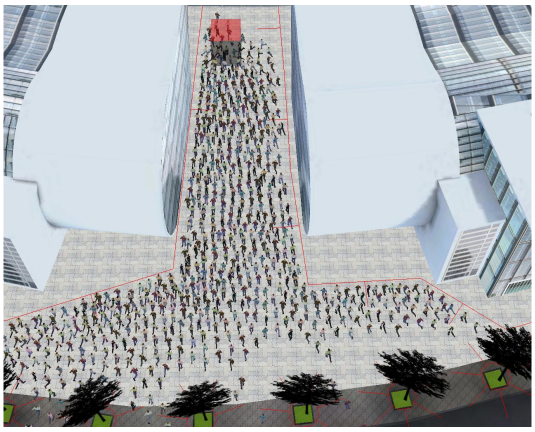
Figure 7.20: Barrier established to cut the street

In both cases described above the user specifies the high level goal only. Once that happens and the task is delegated to a policeman, the agent will use the planner to determine the required course of action in order to be able to fulfill the order. Typically, this involves recruiting one or several car agents, moving to the designated location and establishing the barrier in place, as shown in fig. 7.20.

7.6.3 Technical notes

Apart from the standard collaboration framework as described in the chapter 6, a few modifications were necessitated by the large scale of the scenario ($\approx$ 90 agents – 20 policemen, 20 cars, 20 policeman puppets, 20 car puppets and auxiliary agents, such as the facilitator and planning agent).

First of all, the visualization engine was changed from the in-house developed VHD++ to Delta3D because at the time VHD++ didn't support animation of crowds. Specialized crowd rendering is a necessity if a reasonable performance is to be achieved with many virtual characters being animated at the same time.


(a) Crowd following the leader while avoiding obstacles

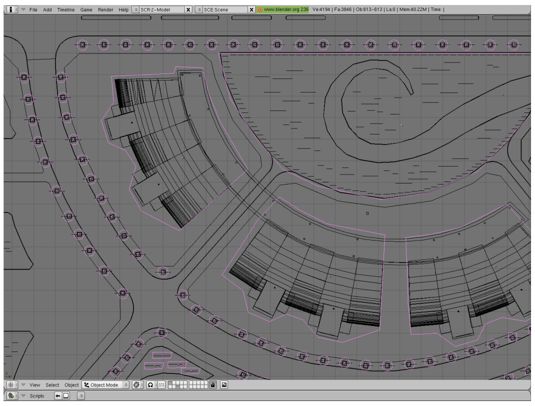

(b) Corresponding obstacle definition in Blender

Figure 7.21: Social forces model for the crowd

The crowd is controlled by a single agent designated as leader. The crowd members follow the leader using the social force field model described by Helbing in [39]. The obstacles were defined using the same tool as for the bar scenario described in section 7.5. Figure 7.21 illustrates the crowd behavior when faced with an obstacle (in red).

The standard CORBA communication layer between VHD++ and the puppets was replaced by an event-based layer using the COS Event Service, as defined by OMG standard. The reason for this change was the requirement of asynchronous communication between the 3D engine and the simulation framework.

With the regular VHD++ based system, the communication is based on blocking function calls – with the consequence that the puppet will be blocked until the virtual character finishes the requested animation. This is not desirable when many virtual characters have to be moved at the same time – they would have to move sequentially, each waiting until the previous one finishes his action. With 40 dynamically simulated agents (20 policemen and 20 cars) such system would be unusable.

Another change concerns the distributed nature of the simulator. The full version with 90 agents is too large to be run on a single machine as the previous studies were (such as the virtual guide described in section 7.2). The system was distributed over three computers, as depicted in figure 7.22. One of the machines used was a high performance Linux cluster, it was used to run the agent framework. The PC workstation powered the visualization (using a large back-projected screen) and the user interface. Finally, a laptop was used to run the gamepad ghost.

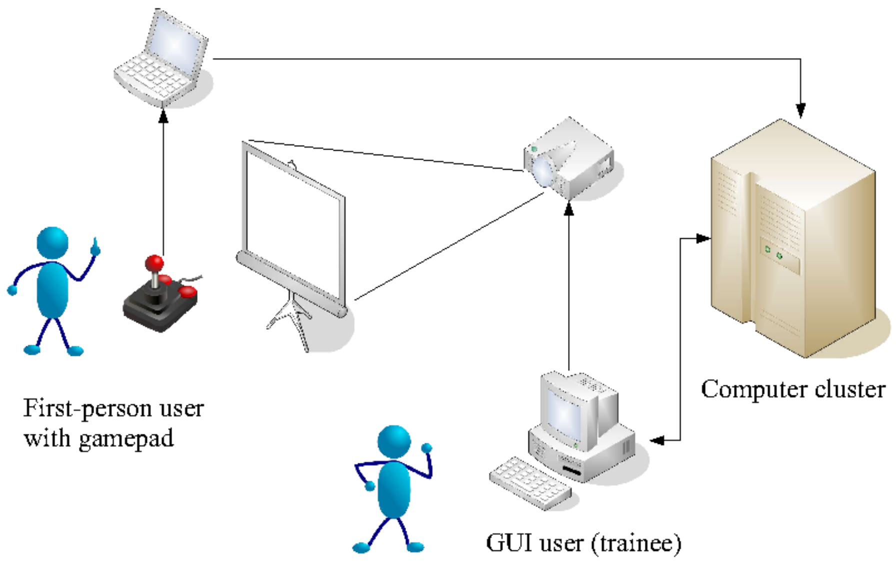


Figure 7.22: System setup for the City Riot simulator

The basic functionality of the collaboration framework used for this scenario was changed only minimally. One notable exception is the use of the multiple planners - the planning agent runs multiple instances of the planner to achieve higher planning throughput when multiple ghosts request planning.

Further exception from the standard framework is the use of the parallelization step to enable out-of-order execution of the tasks (Metric-FF is a total order planner generating sequential plans). The agents are submitting delegated movement tasks to the facilitator in parallel and waiting for all of them to finish at once. The parallelization is necessary because of the large amount of the agents. Moving the larger groups sequentially would take too long and is not realistic neither.

7.6.4 Evaluation

The training simulator is the largest test of the collaborative agent framework described in this thesis. The goal of the study was to validate the framework from several points of view:

- Scalability. The initial tests used only few agents ($\approx$ 10) and the facilitator was perceived as a possible bottleneck for larger simulation.

- Complexity. The previous experiments used only few actions and very small teams (1-2 team members + leader). A more realistic scenario was required to evaluate the impact of added information on both the facilitator and the planner.

- Network transparency. The system was developed from scratch with the focus on network transparency (as provided by CORBA), but it was never truly tested before. All simulations were run on a single machine, because there was no real need for distributed computing.

From the point of view of scalability, the framework performed well. The expected bottleneck caused by the facilitator didn't have a perceivable impact with the ≈ 90 agents used by the scenario. The lightweight design of the facilitator has proven itself as the correct decision.

Of course, if an even larger scenario is desired, it will be necessary to make certain changes. Right now the facilitator spends most of its time in performing the unification and solvable look-ups and the impact of these two rather expensive operations will only grow as new agents are added. One solution would be to switch the storage back-end to a relational database and performing parts of the matching directly in SQL code. Another possibility is a multi-facilitator system, where the work is divided among multiple facilitator instances.

From the complexity point of view, the largest impact is on the amount of data the facilitator has to deal with and the difficulty of the planning required to achieve given goals. The issue of the facilitator performance was discussed above already. The planner is a critical problem as the amount of data and agents grow. Propositional planning has a tendency for explosive complexity growth when new data are added to the initial state or new operators are introduced.

This issue forced also the switch from the Sensory Graphplan to Metric-FF in spite of a significant complication of Metric-FF being a total order planner. Metric-FF is a forward-chaining planner and due to the use of heuristics for guiding the search it is several orders of magnitude faster – as a comparison, Metric-FF was able to solve the Towers of Hanoi with 9 disks in 511 steps (the amount of steps is $2^n - 1$, where n is the number of disks) in under a second. Sensory Graphplan takes more than 20 minutes just to determine that the default amount of steps allowed (10) is not sufficient to solve the problem.

Despite that Metric-FF is one of the fastest planners currently available (according the standard IPC[4] benchmarks for numeric domains), several limits were hit during the development of the scenario. During the attempts to solve the issue of parallelization of the total order plans generated by Metric-FF, one solution would be to move the whole team (e.g. consisting of 5 agents) at once using a single "mass-move" action. Such an approach would elegantly sidestep the problems of out-of-order action execution and ensure the state consistency of the system. Unfortunately, expanding the argument list (8 arguments instead of just 3 – the leader, 5 team member, origin, destination) of the corresponding operator led to complexity explosion in the operator instantiation step because of too many possible permutations and the planner ran out of memory. Another problem was encountered with rooms and links between them – if there are too many ($>\approx 20$ in our case) rooms and links, there will be too many possible instantiations of the `move` operator and the planner either runs very slowly (needs to search a large graph) or runs out of memory.

These issues have to be carefully tested and tuned. To improve the behavior of the planner in cases where there are many irrelevant facts, it would be useful to implement a pre-processing step to filter out the irrelevant information. One such method based on solving a dual problem (transformation from goal state back to initial state) was described in [22].

[4]International planning competition, `http://planning.cis.strath.ac.uk/competition/`

Network transparency was a design feature which was built into the collaboration framework right from the beginning when the decision to use CORBA for inter-agent communication purposes was made. The correctness of this decision was verified when the described simulation was able to run on anything from a single laptop to a high-performance cluster without any modification. The only modification necessary is correct specification of the location of the COS Naming service used by the agents to locate each other on the network. Such flexibility allows easy adaptation of the framework to the complexity of the solved problem.

Another advantage of a network transparent architecture is that implementation of multi-user simulations is very easy – it is sufficient to connect the required software agents (such as a gamepad or a GUI) for different users. No special handling is necessary, with the exception of unique resources such as camera control.

Overall, the described case study was a success validating many design and implementation decisions made during the development of the described collaborative framework. The study proves that the implemented framework and used techniques are usable and useful for solving real-world problems in the simulation and training fields.

Chapter 8

Conclusions

8.1 Summary of the research

This dissertation focused on the problem of human-agent collaboration in virtual environments with the focus on problem solving by propositional planning and the related issues, such as representation of the semantic information about the world and problems of control and controllability.

This work tried to address these issues and focused on several goals:

- Different control modes – using the ghosts & puppets framework enables the application developer to easily provide multiple interaction and control modes, such as first person direct interaction or indirect interaction by issuing orders. The user is able to select a mode of operation which suits him or the task at hand best, contributing to his comfort and efficiency.

  Another advantage is the possibility of direct intervention in case that the otherwise autonomous virtual characters do not behave in desired manner. This feature is of special importance in training and therapeutic applications, where the trainer or therapist have to be in control of the application at all times.

- Task delegation – let the computer do the menial jobs it was designed for. The tasks can be either not interesting but necessary to do or simply too complex for a human user to perform manually. Delegation is a basic element of collaboration between the parties, both human users and autonomous agents.

  The delegation of work to the machine was achieved by a facilitator-based multi-agent framework enabling the human user but also the agents themselves to ask other participants (both humans and computers) to help with solving of a particular task.

- Automatic sub-task solving – in order to be really useful, it is not enough to only delegate the task, but the simulator has to be also able to solve it autonomously. To aid with this task a planning system was implemented.

  The planner together with the task delegation facilities enable the user to shift the complexities of the problem being solved to the autonomous agents. It also frees the user from having to deal with low-level details which the machine can work out automatically.

- Teamwork – many tasks are difficult or not solvable at all without collaboration with others. In order to address this problem, the multi-agent framework was enhanced with explicit teamwork support using delegated actions and team planning to enable collaboration between human users and autonomous agents and between agents themselves.

As a result, a multi-agent collaborative framework was implemented, which tries to address all these goals in a consistent manner. The work was focused on virtual reality simulations using virtual humans with sample applications in therapy, training and elsewhere.

8.2 Summary of the contributions

The main contribution of this work is the new approach to the human user – intelligent agent collaboration and problem solving, based on task delegation, planning and teamwork and focused on the virtual environments.

During the course of this research, several new solutions and techniques were developed and implemented:

- *The ghost & puppets framework* – it permits to exchange and share the control over a virtual character or object dynamically at run-time. This is a new solution of the problem of control over the simulation, which is especially important in training and therapeutic applications. The implemented solution is also useful in case that multiple control modes are required during run-time in the same application.

- *Knowledge and semantic information representation in the virtual environment* – a formal method how to define semantic information for all major components of the virtual reality application was demonstrated. The semantics of the world state, tasks and agents' actions was defined using predicate calculus and STRIPS-like notation.

- *A multi-agent collaborative simulation framework based on task delegation, facilitation and planning* – implements the support for intelligent autonomous agents able to work in teams to solve problems. The focus of this framework is on problem solving and teamwork for both human – intelligent agent and agent – agent cases.

- *Delegated actions in standard STRIPS-like planners* – they allow the planner to reason about teams and teamwork by describing the expected behavior of the team members. Together with the Contract Net-derived protocol for team forming, delegated actions form a basis for collaborative activity between the agents in the described framework.

- *Object-specific planning* – the concept of smart objects was extended to permit reasoning about them by including object-specific high level semantic information about the object's properties, state and actions possible to perform with it.

8.3 Future work

The teamwork between autonomous agents and more importantly between a human user and an autonomous agent is a very complex topic and there are still many problems left to address. Some of the issues identified during the course of this work will be presented in the following sections.

8.3.1 State consistency

Consistency of the semantic information about the state of the virtual world is major problem. Maintaining it using only the declared effects of the actions fails to take into account events which happened without the direct influence of the agent (for example because of result of some physical simulation, such as falling blocks). Agents' sensing mitigates this problem to some extent, however it is still an issue in the case of agent/human user having total observability – the user will have incorrect information about the world state.

More research needs to be done in order to make the bond between the semantic information and the geometric/physical representation of the objects closer. Ideally, each object should be able to update the world state automatically, whenever its own state changes.

8.3.2 Sub-teams

The current implementation of the simulation framework permits the agent to be a member of only a single team at a time. However, it is frequently useful to let the team member become a leader of its own sub-team focused on solving the task that it got assigned. This feature is feasible as a straightforward extension of the existing implementation.

8.3.3 Agent autonomy versus controllability

The issue of agent's autonomy has to be carefully balanced against the controllability of the scenario. In the implemented case studies, the focus was on the controllability and the agents have very little autonomous behavior – they mostly wait for delegated tasks or perform some simple scripted activities until told otherwise.

The problem with agents which are autonomous to a higher degree is the interplay between their own desires (assuming the common BDI model) and the higher level goals delegated to them (roles assigned to them in the collaboration process). More research is needed to establish good balance between the two extremes – fully autonomous agent ignoring the delegated tasks and fully externally controlled agent with minimal autonomy. A first attempt to address this problem was made by using different priorities for incoming delegated tasks and internally generated tasks (intentions to satisfy own desires), however that is very crude solution.

8.3.4 Facilitator improvements

The facilitator plays a central role in the multi-agent collaborative framework described in this dissertation. There are several problems with the current implementation, mainly concerning the scalability of the current implementation.

The current facilitator has bottlenecks while matching the incoming requests against its knowledge base because of the rather inefficient organization of the data. In case of a simulation system with many agents and a lot of state being stored, another back-end supporting efficient matching would be necessary. A most logical solution would be a port to a relational database.

Another possible improvement for large deployments is inclusion of the support for multiple facilitators and transparent data replication/synchronization between them. Alternatively, support for data partitioning and deep queries between facilitators would have to be implemented.

Appendix A

Extended smart objects

A.1 Jukebox from the bar environment

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<vhdHObjectProperty name = "SOjukeBox">
<hobjObjectoid name = "dancefloor-jukebox">
        <hobjMatrix for = "worldTransform">
                1.000000 0.000000 0.000000 0.000000
                0.000000 1.000000 0.000000 0.000000
                0.000000 0.000000 1.000000 0.000000
        0.000000 0.000000 0.000000 1.000000
        </hobjMatrix>

        <hobjGroup name = "JukeChooser">
                <hobjMatrix for = "defaultTransform">
                        0.000000 0.000000 -1.000000 0.000000
                        0.000000 1.000000 0.000000 0.000000
                        1.000000 0.000000 0.000000 0.000000
                        -8.761451 0.944456 3.338238 1.000000
                </hobjMatrix>

                <hobjVisualGeometry name = "jukeChooser">
                        <hobjMatrix for = "offsetTransform">
                                0.010000 0.000000 0.000000 0.000000
                                0.000000 0.010000 0.000000 0.000000
                                0.000000 0.000000 0.010000 0.000000
                                0.000000 0.000000 0.000000 1.000000
                        </hobjMatrix>

                        <hobjFile>jukeBox.osg</hobjFile>
                </hobjVisualGeometry>

                <hobjAttributeSet name = "jukeRHand" annotation = "hand">
                        <hobjBool for = "left">
                                false
                        </hobjBool>
                        <hobjMatrix for = "transform">
                                -0.986668 -0.154289 -0.051783 0.000000
                                -0.098919 0.315876 0.943630 0.000000
                                -0.129234 0.936172 -0.326927 0.000000
                                0.236765 0.139402 0.190024 1.000000
                        </hobjMatrix>
                        <hobjQuat for = "pinky0"> 9.1709e-009 4.45565e-010 2.54339e-009 1 </hobjQuat>
                        <hobjQuat for = "ring0"> 9.1709e-009 4.45565e-010 2.54339e-009 1 </hobjQuat>
                        <hobjQuat for = "middle0"> 9.1709e-009 4.45565e-010 2.54339e-009 1 </hobjQuat>
                        <hobjQuat for = "index0"> 9.1709e-009 4.45565e-010 2.54339e-009 1 </hobjQuat>
                        <hobjQuat for = "thumb1"> 7.59119e-009 0.188775 4.88034e-009 0.98202 </hobjQuat>
                        <hobjQuat for = "pinky1"> 9.38865e-009 -8.39039e-010 0.513431 0.858131 </hobjQuat>
                        <hobjQuat for = "pinky2"> -5.00326e-008 -4.95325e-008 0.625986 0.779834 </hobjQuat>
                        <hobjQuat for = "pinky3"> 2.86096e-009 -4.76598e-009 0.558285 0.829649 </hobjQuat>
                        <hobjQuat for = "ring1"> -1.25456e-007 -5.46972e-008 0.370503 0.928831 </hobjQuat>
                        <hobjQuat for = "ring2"> 2.38569e-009 -2.12607e-009 0.561404 0.827542 </hobjQuat>
                        <hobjQuat for = "ring3"> -6.6419e-008 -4.3602e-008 0.558285 0.829649 </hobjQuat>
                        <hobjQuat for = "middle1"> 1.4213e-008 -7.24679e-010 0.192504 0.981296 </hobjQuat>
                        <hobjQuat for = "middle2"> -6.44407e-008 -3.01425e-008 0.353089 0.93559 </hobjQuat>
                        <hobjQuat for = "middle3"> -6.67108e-008 -2.23531e-008 0.401332 0.915933 </hobjQuat>
                        <hobjQuat for = "index1"> -5.07543e-009 5.97495e-011 3.23166e-009 1 </hobjQuat>
                        <hobjQuat for = "index2"> 9.1709e-009 5.55434e-009 -9.86178e-010 1 </hobjQuat>
                        <hobjQuat for = "index3"> 9.1709e-009 5.55434e-009 -9.86178e-010 1 </hobjQuat>
                        <hobjQuat for = "thumb2"> 0.167034 8.85852e-009 -1.70661e-009 0.985951 </hobjQuat>
                        <hobjQuat for = "thumb3"> 0.122157 0.0170392 -0.137094 0.982849 </hobjQuat>
                </hobjAttributeSet>
        <hobjAttributeSet name = "standPosition" annotation = "position">
            <hobjPoint for = "position">
                0.0 0.0 0.7
            </hobjPoint>

            <hobjVector for = "direction">
                0.0 0.0 -1.0
            </hobjVector>
        </hobjAttributeSet>

                <hobjAttributeSet name = "jukePos0" annotation = "position">
                        <hobjPoint for = "position">
                                -0.100873 -0.944456 0.488811
                        </hobjPoint>
                </hobjAttributeSet>
                <hobjAttributeSet name = "jukePos1" annotation = "position">
                        <hobjPoint for = "position">
                                -0.063293 -0.944456 2.342339
                        </hobjPoint>
                </hobjAttributeSet>
                <hobjAttributeSet name = "jukePos2" annotation = "position">
                        <hobjPoint for = "position">
                                1.436546 -0.944456 2.544468
                        </hobjPoint>
                </hobjAttributeSet>
                <hobjAttributeSet name = "jukeVector" annotation = "vector">
                        <hobjPoint for = "position">
                                -0.100873 -0.944456 0.488811
                        </hobjPoint>
                        <hobjVector for = "direction">
                                -1.000000 0.000000 0.000000
                        </hobjVector>
                </hobjAttributeSet>

        <hobjAttributeSet annotation="" name="object-properties">
                <hobjText for="predicates">
                        [('jukebox', 'dancefloor-jukebox'),
                         ('machine', 'dancefloor-jukebox')]
                </hobjText>

                <hobjText for="functions">
                        {'(dancefloor-jukebox-FOO)':0}
                </hobjText>
        </hobjAttributeSet>

        <hobjAttributeSet name = "operator-a-power-up-jukebox" annotation = "action">
<hobjText for="functor">
import time
import mathutils
import unifier
import SOKOBAN
import action_functor

class a_power_up_jukebox(action_functor.GenericActionFunctor):
    '''
    Actions are integrated into functor classes, i.e classes that implement the
    __callable__ method and can therefore be called as functions
    this way in addition to executing the action we can also call
    other methods such as verify_preconds or others
    '''
    def __init__(self, human):
        action_functor.GenericActionFunctor.__init__(self, human)
        self.action_name = 'power-up-jukebox'
        self.arguments = ('?who', '?what')
        self.vars = []

        self.predicates = [('near', '?who', '?what'),
                           ('agent', '?who'),
                           ('busy', '?who'),
                           ('powered-on', '?what'),
                           ('machine', '?what'),
                           ('jukebox', '?what')]

        self.preconds = ('and',
                         ('near', '?who', '?what'),
                         ('agent', '?who'),
                         ('machine', '?what'),
                         ('not', ('busy', '?who')),
                         ('not', ('powered-on', '?what')))

        self.effects = ('powered-on', '?what')

    def verifyPreconds(self, *args):
        '''
        replaces all occurences of "?from" and "?to" with
        actual locations and verifies whether the action is solvable
        '''
        preconditions = unifier.apply_subs({'?who':args[0],
                                            '?what':args[1]}, self.preconds)
        return self.human._check_solvable(preconditions)


    def __call__(self, *args):
        ''' toggle the device on
        '''

        self.logger.info('power-up-jukebox called, args = ' + repr(args))

        # check preconds
        # these have to be invariants

        who = args[0]
        machine = args[1]

        if self.human.human_name != who:
            return ('error', ('badagent', self.human.human_name, who))

        if self.verifyPreconds(*args):
            # push the button
            try:
                machine_puppet = self.human.chelper.openObject(machine)
                machine_puppet.possess(self.human.ghost_name, 0)
                machine_puppet.performGenericAction(self.human.ghost_name, \
                                                    'power-up-jukebox', \
                                                    [self.human.human_name, machine])
                machine_puppet.unpossess(self.human.ghost_name,0)

            except SOKOBAN.ActionFailed, err:
                # oops, failure
                code = eval(err.reason)
                if code[0] == 'nopath':
                    self.logger.error('path not found between to ' + repr(pos))
                elif code[0] == 'obstacle':
                    self.logger.error('obstacle at : ' + repr(code[1]))
                else:
                    self.logger.error('unknown problem occurred')

                # try to salvage the state, because we are maybe not in the original place anymore
                # nothing to do here
                return ('error', code)
            else:
                # update state
                # turning machine on
                self.human._declare_add_effects([('powered-on', machine)])
                self.human.logger.info('power-up-jukebox - machine turned on' + str(machine) )

                return ('success', )
        else:
            self.logger.error('Preconditions for power-up-machine not met!')
            return ('error', ('precond',))

</hobjText>
<hobjText for="animation_script">
sys.path.append('/home/janoc/src/VHD/VHDPP_TEST/py')
import time
from libhobject import *
from pyvhdHObjectService import *
from pyvhdHAGENTService import *
from pyvhdPythonService import *

eH = PythonEventHandler()
oS = hobjectService
ag = hagentService

human = Human(ARGS[0])

jukebox = ObjectoidPtr(oS.getHObjectByName(ARGS[1]))

human.walk(asAttributeSet(jukebox.getNodeByName("standPosition")), Human.Stop)
eH.subscribe(WalkReachedEventClass(human))
eH.waitForEvents()

#fix orientation
q = vhdQuaternion()
q.fromAngleAxis(-3.14/2.0, vhdVector3(0,1,0))
ag.setRotation(ARGS[0], q)

rh = asAttributeSet(jukebox.getNodeByName("jukeRHand"))
human.grasp(rh)
human.reach(human.RightArm, rh.lookupUpdated("transform"))

time.sleep(2)

human.reset()

</hobjText>
                </hobjAttributeSet>
                <hobjAttributeSet name = "operator-a-power-down-jukebox" annotation = "action">
<hobjText for="functor">
import time
import mathutils
import unifier
import SOKOBAN
import action_functor

class a_power_down_jukebox(action_functor.GenericActionFunctor):
    '''
    Actions are integrated into functor classes, i.e classes that implement the
    __callable__ method and can therefore be called as functions
    this way in addition to executing the action we can also call
    other methods such as verify_preconds or others
    '''
    def __init__(self, human):
        action_functor.GenericActionFunctor.__init__(self, human)
        self.action_name = 'power-down-jukebox'
        self.arguments = ('?who', '?what')
        self.vars = []

        self.predicates = [('near', '?who', '?what'),
                           ('agent', '?who'),
                           ('busy', '?who'),
                           ('powered-on', '?what'),
                           ('machine', '?what'),
                           ('jukebox', '?what')]

        self.preconds = ('and',
                         ('near', '?who', '?what'),
                         ('agent', '?who'),
                         ('machine', '?what'),
                         ('not', ('busy', '?who')),
                         ('powered-on', '?what'))

        self.effects = ('not', ('powered-on', '?what'))

    def verifyPreconds(self, *args):
        '''
        replaces all occurences of "?from" and "?to" with
        actual locations and verifies whether the action is solvable
        '''
        preconditions = unifier.apply_subs({'?who':args[0],
                                            '?what':args[1]}, self.preconds)

        return self.human._check_solvable(preconditions)


    def __call__(self, *args):
        ''' toggle the device on
        '''

        self.logger.info('power-down-jukebox called, args = ' + repr(args))

        # check preconds
        # these have to be invariants

        who = args[0]
        machine = args[1]

        if self.human.human_name != who:
            return ('error', ('badagent', self.human.human_name, who))

        if self.verifyPreconds(*args):
            # push the button
            try:
                machine_puppet = self.human.chelper.openObject(machine)
                machine_puppet.possess(self.human.ghost_name, 0)
                machine_puppet.performGenericAction(self.human.ghost_name, \
                                                    'power-down-jukebox', \
                                                    [self.human.human_name, machine])
                machine_puppet.unpossess(self.human.ghost_name,0)

            except SOKOBAN.ActionFailed, err:
                # oops, failure
                code = eval(err.reason)
                if code[0] == 'nopath':
                    self.logger.error('path not found between to ' + repr(pos))
                elif code[0] == 'obstacle':
                    self.logger.error('obstacle at : ' + repr(code[1]))
                else:
                    self.logger.error('unknown problem occurred')

                # try to salvage the state, because we are maybe not in the original place anymore
                # nothing to do here
                return ('error', code)
            else:
                # update state
                # turning machine off
                self.human._undeclare_del_effects([('powered-on', machine)])
                self.human.logger.info('power-down-jukebox - machine turned off' + str(machine) )

                return ('success', )
        else:
            self.logger.error('Preconditions for power-down-machine not met!')
            return ('error', ('precond',))

</hobjText>
<hobjText for="animation_script">
sys.path.append('/home/janoc/src/VHD/VHDPP_TEST/py')
import time
from libhobject import *
from pyvhdHObjectService import *

eH = PythonEventHandler()
oS = hobjectService

human = Human(ARGS[0])

jukebox = ObjectoidPtr(oS.getHObjectByName(ARGS[1]))

human.walk(asAttributeSet(jukebox.getNodeByName("standPosition")), Human.Stop)
eH.subscribe(WalkReachedEventClass(human))
eH.waitForEvents()

#fix orientation
q = vhdQuaternion()
q.fromAngleAxis(-3.14/2.0, vhdVector3(0,1,0))
ag.setRotation(ARGS[0], q)

rh = asAttributeSet(jukebox.getNodeByName("jukeRHand"))
human.grasp(rh)
human.reach(human.RightArm, rh.lookupUpdated("transform"))

time.sleep(2)
human.reset()

</hobjText>
                </hobjAttributeSet>

        </hobjGroup>
</hobjObjectoid>
</vhdHObjectProperty>
```

Appendix B

Example actions

B.1 move action

```
######
##### Functor class move.
##### Implements the action of moving
##### Created by Miguel Garcia Arribas (miguel.garcia@epfl.ch)
##### 14/12/2004
#####

import pickle
import mathutils
import unifier
import SOKOBAN
import action_functor

class a_move(action_functor.GenericActionFunctor):
    '''
    Actions are integrated into functor classes, i.e classes that implement the
    __callable__ method and can therefore be called as functions
    this way in addition to executing the action we can also call
    other methods such as verify_preconds or others
    '''
    def __init__(self, human):
        action_functor.GenericActionFunctor.__init__(self, human)
        self.action_name = 'move'
        self.arguments = ('?who', '?from','?to')

        self.preconds = ('and',  ('at', '?who', '?from'),
                          ('not', ('=', '?from', '?to')),
                          ('place', '?from'),
                          ('place', '?to'),
                          ('agent', '?who'),
                          ('not', ('busy', '?who')),
                          ('connected', '?from', '?to'))

        self.effects = ('and',
                          ('at', '?who', '?to'),
                          ('not', ('at', '?who', '?from')))

        self.predicates = [('at', '?who', '?from'),
                            ('place', '?from'),
                            ('agent', '?who'),
                            ('busy', '?who'),
                            ('connected', '?from', '?to')]

    def verifyPreconds(self, *args):
        '''
        replaces all occurences of "?from" and "?to" with
        actual locations and verifies whether the action is performable
        '''
        preconditions = unifier.apply_subs({'?who':args[0],
                                            '?from':args[1],
                                            '?to':args[2]}, self.preconds)

        return self.human._check_solvable(preconditions)


    def __call__(self, *args):
        ''' move agent from one place to another
        '''
        #self.human.logger.debug(self.writeLisp(True))
        # check preconds
        self.human.logger.debug('called a_move' + repr(args))

        who    = args[0]
        p_from = args[1]
        p_to   = args[2]

        if self.human.human_name != who:
            return ('error', ('badagent', self.human.human_name, who))

        if self.verifyPreconds(*args):
            # move around to the closest exit point in the other room
            current_pos = self.human.human_puppet.reportPos()

            # get exits
            solvable = ('exits_for', p_to)
            res = self.human._wait_for_reply(self.human.fac.solve(pickle.dumps(solvable),
                                                       pickle.dumps({'no_results':1}), self.human.me), 60)

            if res[0][0] == 'success':
                exits = res[0][1]
                #p_to_pos = mathutils.getClosest((current_pos.x, current_pos.y), exits)
                p_to_pos = random.choice(exits);
            else:
                self.human.logger.error('Invalid destination ' + p_to + ' for move !')
                return ('error', ('badplace', p_to))

            # failure is not an option, however the puppet can easily mess up
            # since it has no brains ...
            try:
                self.human.human_puppet.goto(self.human.ghost_name, p_to_pos[0], p_to_pos[1])

            except SOKOBAN.ActionFailed, err:
                # oops, failure
                code = eval(err.reason)
                if code[0] == 'nopath':
                    self.human.logger.error('path not found between ' + p_from + ' and ' + p_to)
                elif code[0] == 'obstacle':
                    self.human.logger.error('obstacle at : ' + repr(code[1]))
                else:
                    self.human.logger.error('unknown problem occurred')

                # try to salvage the state, because we are maybe not in the original place anymore
                self.human._undeclare_old_position(self.human.human_name)
                #self._undeclare_del_effects([('at', self.human_name, p_from)])

                self.human.current_room = self.human._which_room(self.human.human_puppet.reportPos())
                self.human._declare_add_effects([('at', self.human.human_name, self.human.current_room)])
                return ('error', code)
            else:
                # update state
                self.human._undeclare_old_position(self.human.human_name)
                #self._undeclare_del_effects([('at', self.human_name, p_from)])
                self.human._declare_add_effects([('at', self.human.human_name, p_to)])
                self.human.current_room = p_to
                return ('success',)
        else:
            self.human.logger.error('Preconditions for move not met!')
            return ('error', ('precond',))
```

B.2 delegated-move action

```
import mathutils
import unifier
import SOKOBAN
import action_functor

class a_delegated_move(action_functor.GenericActionFunctor):
    '''
    Actions are integrated into functor classes, i.e classes that implement the
    __callable__ method and can therefore be called as functions
    this way in addition to executing the action we can also call
    other methods such as veryfy_preconds or others
    '''
    def __init__(self, human):
        action_functor.GenericActionFunctor.__init__(self, human)
        self.action_name = 'delegated-move'
        self.arguments = ('?who', '?from','?to')

        self.preconds = ('and',
                         ('or', ('at', '?who', '?from'),
                                ('=', '?from', 'anywhere')),
                         ('not', ('=', '?from', '?to')),
                         ('place', '?from'),
                         ('place', '?to'),
                         ('agent', '?who'),
                         ('not', ('busy', '?who')))

        self.effects = ('and',
                         ('at', '?who', '?to'),
                         ('not', ('at', '?who', '?from'))
                         )

        self.predicates = [('at', '?who', '?from'),
                           ('place', '?from'),
                           ('agent', '?who'),
                           ('connected', '?from', '?to')]

    def verifyPreconds(self, *args):
        '''
        replaces all occurences of "?from" and "?to" with
        actual locations and verifies whether the action is solvable
        '''
        preconditions = unifier.apply_subs({'?who':args[0],
                                            '?from':args[1],
                                            '?to':args[2]}, self.preconds)

        return self.human._check_solvable(preconditions)


    def __call__(self, *args):
        ''' move agent from one place to another
        '''
        #self.human.logger.debug(self.writeLisp(True))

        # check preconds
        self.human.logger.debug('called a_delegated_move' + repr(args))

        who    = args[0]
        p_from = args[1]
        p_to   = args[2]

        # CAUTION!
        # this is not a bug, the lisp delegate-move and this
        # function are not the same thing. The one in lisp is from the
        # point of view of a team leader giving order, this one here is from
        # the point of view of the *TEAMMATE* receiving it. Therefore these two
        # conditions are not checked here:
        # ('not', ('=', '?who', 'self')),
        # ('teamleader', 'self')

        if self.human.human_name != who:
            result = ('error', ('badagent', self.human.human_name, who))
        else:
            if self.verifyPreconds(*args):
                if self.human._which_room(self.human.human_puppet.reportPos()) == p_to:
                    # he's already there, so there's no need to ask the planner to do anything.
                    # in fact if we do, it will return 'none' and the whole planning will fail.
                    result = ('success', )
                else:
                    # he's somewhere else, so it's safe to call the planner
                    # ask the planner to move us to the destination
                    result = self.human.action_table['plan-and-execute'](self.human.human_name, \
                                                                        ('at', self.human.human_name, p_to))
            else:
                self.human.logger.error('Preconditions for delegated-move not met!')
                result = ('error', ('precond',))

        return result
```

Appendix C

“Cut street” order from the “City riot” application

A real example from the “City riot” application. The user (trainee) orders his policemen to block a street using their vehicles. The team-forming and the usage of delegated actions is clearly visible. The planner also decides that it is faster to take the team leader to the destination by car than by walking and he uses one of the cars for transportation. The problem description part shows the current beliefs of the team leader as well.

C.1 Domain description (operators)

```
(define (domain door) (:requirements :strips :equality)

(:predicates (self ?foo)
(toilet ?what)
(ordered ?who ?what)
(thirsty ?who)
(agent ?who)
(guarding ?a ?slot)
(at ?who ?what)
(human ?who)
(car-covered ?slot)
(busy ?who)
(named-point ?what)
(teammember ?who)
(open ?door)
(consumable ?what)
(near ?who ?what)
(covered ?slot)
(powered-on ?what)
(teamleader ?who)
(door ?door)
(joining-team ?who)
(fire ?slot)
(strategic-point ?arg_0 ?arg_1)
(barman ?who)
(connected ?from ?to)
(protoagent ?who)
(being-guided ?who)
(dance-place ?what)
(uncertain ?door)
(have-consumable ?who ?what)
(policeman ?who)
(waiting-for-guide ?who)
(doorway ?door ?side_a ?side_b)
(place ?what)
(guiding ?who)
(car ?who)
)
(:functions (THIRST)
(FEELFULL)
(TEAMSIZE)
)
(:action dance
 :parameters (?who ?jukebox ?where)
 :precondition  (and (agent ?who) (place ?where)
                     (dance-place ?where) (at ?who ?where)
                     (powered-on ?jukebox) (= ?who self))
 :effect  (increase (THIRST) 20)
)
(:action declare-slot-car-protected
 :parameters (?who ?a ?slot)
 :vars (?place)
 :precondition  (and (agent ?who) (agent ?a) (policeman ?who)
                     (car ?a) (place ?place) (at ?a ?place)
                     (strategic-point ?place ?slot) (named-point ?slot)
                     (not (fire ?place)) (guarding ?a ?slot) (= ?who self))
 :effect  (car-covered ?slot)
)
(:action delegated-protect-building-slot
 :parameters (?who ?slot)
 :vars (?place)
 :precondition  (and (agent ?who) (place ?place) (at ?who ?place)
                     (strategic-point ?place ?slot) (named-point ?slot)
                     (not (fire ?place)) (not (busy ?who))
                     (not (= ?who self)) (teamleader self))
 :effect  (and (guarding ?who ?slot) (busy ?who))
)
(:action delegated-bring-order
 :parameters (?who ?to_whom ?what)
 :precondition  (and (ordered ?to_whom ?what) (agent ?who)
                     (agent ?to_whom) (barman ?who) (consumable ?what)
                     (not (= ?to_whom ?who)) (not (= ?who self)) (teamleader self))
 :effect  (have-consumable ?to_whom ?what)
)
(:action delegated-drive-agent
 :parameters (?who ?whom ?from_vehicle ?from_policeman ?to)
 :precondition  (and (agent ?who) (agent ?whom) (car ?who)
                     (policeman ?whom) (place ?from_policeman)
                     (place ?to) (place ?from_vehicle)
                     (at ?whom ?from_policeman) (at ?who ?from_vehicle)
                     (not (= ?from_policeman ?to)) (not (busy ?who))
                     (not (busy ?whom)) (not (= ?who self)) (teamleader self))
 :effect  (and (at ?whom ?to) (not (at ?whom ?from_policeman))
               (at ?who ?to) (not (at ?who ?from_vehicle)))
)
(:action quench-thirst
 :parameters (?who)
 :precondition  (and (thirsty ?who) (agent ?who) (<= (THIRST) 20) (= ?who self))
 :effect  (not (thirsty ?who))
)
(:action declare-slot-protected
 :parameters (?who ?a ?slot)
 :vars (?place)
 :precondition  (and (agent ?who) (agent ?a) (policeman ?who)
                     (place ?place) (at ?a ?place) (strategic-point ?place ?slot)
                     (named-point ?slot) (not (fire ?place))
                     (guarding ?a ?slot) (= ?who self))
 :effect  (covered ?slot)
)
(:action reject-teammate
 :parameters (?who)
 :precondition  (and (joining-team ?who) (agent ?who) (= ?who self))
 :effect  (not (joining-team ?who))
)
(:action delegated-toggle-door
 :parameters (?who ?door)
 :vars (?side_a ?side_b)
 :precondition  (and (agent ?who) (door ?door) (doorway ?door ?side_a ?side_b)
                     (not (= ?who self)) (teamleader self))
 :effect  (and (when (open ?door) (and (not (open ?door))
                     (not (connected ?side_a ?side_b)) (not (connected ?side_b ?side_a))))
               (when (not (open ?door)) (and (open ?door)
                     (connected ?side_a ?side_b) (connected ?side_b ?side_a))))
)
(:action move
 :parameters (?who ?from ?to)
 :precondition  (and (at ?who ?from) (not (= ?from ?to)) (place ?from)
                     (place ?to) (agent ?who) (not (busy ?who))
                     (connected ?from ?to) (= ?who self))
 :effect  (and (at ?who ?to) (not (at ?who ?from)))
)
(:action delegated-get-close
 :parameters (?who ?what)
 :vars (?place)
 :precondition  (and (at ?who ?place) (at ?what ?place)
                     (agent ?who) (not (= ?who self)) (teamleader self))
 :effect  (near ?who ?what)
)
(:action disband-team
 :parameters (?who ?protoagent)
 :precondition  (and (not (= ?who ?protoagent))
                     (agent ?who) (agent ?protoagent)
                     (teamleader ?who) (= ?who self))
 :effect  (and (decrease (TEAMSIZE) 1) (protoagent ?protoagent)
               (not (agent ?protoagent)) (when (= (TEAMSIZE) 1)
               (not (teamleader ?who))))
)
(:action toggle-door
 :parameters (?who ?door)
 :vars (?side_a ?side_b)
 :precondition  (and (doorway ?door ?side_a ?side_b)
                     (or (at ?who ?side_a) (at ?who ?side_b))
                         (agent ?who) (door ?door) (place ?side_a)
                         (place ?side_b) (not (busy ?who)) (= ?who self))
 :effect  (and (when (open ?door) (and (not (open ?door))
                                       (not (connected ?side_a ?side_b))
                                       (not (connected ?side_b ?side_a))))
               (when (not (open ?door)) (and (open ?door)
                                            (connected ?side_a ?side_b)
                                            (connected ?side_b ?side_a))))
)
(:action drink
 :parameters (?who ?what)
 :precondition  (and (agent ?who) (consumable ?what)
                     (have-consumable ?who ?what) (= ?who self))
 :effect  (and (decrease (THIRST) 40) (increase (FEELFULL) 40)
               (not (have-consumable ?who ?what)))
)
(:action delegated-move
 :parameters (?who ?from ?to)
 :precondition  (and (or (at ?who ?from) (= ?from anywhere))
                         (not (= ?from ?to)) (place ?from)
                         (place ?to) (agent ?who) (not (busy ?who))
                         (not (= ?who self)) (teamleader self))
 :effect  (and (at ?who ?to) (not (at ?who ?from)))
)
(:action goto-wc
 :parameters (?who ?wc)
 :vars (?to)
 :precondition  (and (at ?wc ?to) (toilet ?wc)
                     (place ?to) (agent ?who) (not (busy ?who)) (= ?who self))
 :effect  (assign (FEELFULL) 0)
)
(:action delegated-car-protect-building-slot
 :parameters (?who ?slot)
 :vars (?place)
 :precondition  (and (agent ?who) (car ?who) (place ?place)
                     (at ?who ?place) (strategic-point ?place ?slot)
                     (named-point ?slot) (not (fire ?place))
                     (not (busy ?who)) (not (= ?who self)) (teamleader self))
 :effect  (and (guarding ?who ?slot) (busy ?who))
)
(:action leave-as-teammate
 :parameters (?who ?teamleader)
 :precondition  (and (agent ?who) (teammember ?who) (= ?who self))
 :effect  (and (not (teammember ?who)) (not (teamleader ?teamleader)))
)
(:action delegated-take-order
 :parameters (?who ?customer ?what)
 :vars (?place)
 :precondition  (and (agent ?who) (agent ?customer)
                     (barman ?who) (consumable ?what) (place ?place)
                     (at ?who ?place) (at ?customer ?place)
                     (not (= ?customer ?who)) (not (= ?who self)) (teamleader self))
 :effect  (ordered ?customer ?what)
)
(:action commit-to-team
 :parameters (?who ?teamleader)
 :precondition  (and (joining-team ?who) (agent ?who) (= ?who self))
 :effect  (and (teammember ?who) (teamleader ?teamleader) (not (joining-team ?who)))
)
(:action join-as-teammate
 :parameters (?who)
 :precondition  (and (not (busy ?who))
                     (not (joining-team ?who))
                     (agent ?who) (not (teammember ?who)) (= ?who self))
 :effect  (joining-team ?who)
)
(:action recruit-help
 :parameters (?who ?protoagent)
 :precondition  (and (protoagent ?protoagent) (agent ?who) (= ?who self))
 :effect  (and (increase (TEAMSIZE) 1)
               (agent ?protoagent) (teamleader ?who) (not (protoagent ?protoagent)))
)
(:action get-close
 :parameters (?who ?what)
 :vars (?place)
 :precondition  (and (at ?what ?place) (at ?who ?place) (agent ?who) (= ?who self))
 :effect  (near ?who ?what)
)
)
```

C.2 Problem description (initial state, beliefs, goal specification)

```
(define (problem door)
(:domain door)
(:objects
 Leader4
 a1
 a10
 a2
 a3
 a4
 a5
 a6
 a7
 a8
 a9
 anywhere
 field
 lakebuilding
 lakecross1
 lakecross1-slot1
 lakecross1-slot2
 lakecross1-slot3
 lakecross1-slot4
 lakecross3
 lakecross3-slot1
 lakecross3-slot2
 lakecross3-slot3
 lakecross3-slot4
 lakepark
 mainavenue
 parking
 parkingbuilding
 policecross1
 policecross1-slot1
 policecross1-slot2
 policecross1-slot3
 policecross1-slot4
 policecross1-slot5
 policecross1-slot6
 policecross2
 policecross2-slot1
 policecross2-slot2
 policecross2-slot3
 policecross3
 policecross3-slot1
 policecross3-slot2
 policecross3-slot3
 policecross3-slot4
 policecross4
 policecross4-slot1
 policecross4-slot2
 policecross4-slot3
 policecross4-slot4
 policestation
 room1
 room2
 room3
 room4
 room5
 room6
 sidelake
 sidelakestreet
 tvtower
 self
)
(:init
(= (THIRST) 40 )
(= (FEELFULL) 0 )
(= (TEAMSIZE) 0 )
(agent self)
(at self room2)
(at a1 anywhere)
(at a10 anywhere)
(at a2 anywhere)
(at a3 anywhere)
(at a4 anywhere)
(at a5 anywhere)
(at a6 anywhere)
(at a7 anywhere)
(at a8 anywhere)
(at a9 anywhere)
(car a1)
(car a10)
(car a2)
(car a3)
(car a4)
(car a5)
(car a6)
```

```
(car a7)
(car a8)
(car a9)
(connected anywhere field)
(connected anywhere lakebuilding)
(connected anywhere lakecross1)
(connected anywhere lakecross3)
(connected anywhere lakepark)
(connected anywhere mainavenue)
(connected anywhere parking)
(connected anywhere parkingbuilding)
(connected anywhere policecross1)
(connected anywhere policecross2)
(connected anywhere policecross3)
(connected anywhere policecross4)
(connected anywhere policestation)
(connected anywhere room1)
(connected anywhere room2)
(connected anywhere room3)
(connected anywhere room4)
(connected anywhere room5)
(connected anywhere room6)
(connected anywhere sidelake)
(connected anywhere sidelakestreet)
(connected anywhere tvtower)
(connected field room1)
(connected lakebuilding lakecross3)
(connected lakebuilding lakepark)
(connected lakebuilding parkingbuilding)
(connected lakebuilding room5)
(connected lakebuilding room6)
(connected lakecross1 parking)
(connected lakecross1 sidelake)
(connected lakecross3 lakebuilding)
(connected lakecross3 lakepark)
(connected lakecross3 room4)
(connected lakepark lakebuilding)
(connected lakepark lakecross3)
(connected lakepark mainavenue)
(connected lakepark parking)
(connected lakepark room4)
(connected lakepark sidelake)
(connected lakepark sidelakestreet)
(connected mainavenue lakepark)
(connected mainavenue room4)
(connected parking lakecross1)
(connected parking lakepark)
(connected parking parkingbuilding)
(connected parking room2)
(connected parking sidelake)
(connected parking sidelakestreet)
(connected parkingbuilding lakebuilding)
(connected parkingbuilding parking)
(connected parkingbuilding room1)
(connected parkingbuilding room2)
(connected policecross1 policestation)
(connected policecross2 policestation)
(connected policecross3 policestation)
(connected policecross3 sidelake)
(connected policecross4 policestation)
(connected policecross4 sidelake)
(connected policestation policecross1)
(connected policestation policecross2)
(connected policestation policecross3)
(connected policestation policecross4)
(connected policestation sidelake)
(connected room1 field)
(connected room1 parkingbuilding)
(connected room1 room2)
(connected room1 room6)
(connected room2 parking)
(connected room2 parkingbuilding)
(connected room2 room1)
(connected room2 sidelakestreet)
(connected room3 room4)
(connected room3 room5)
(connected room3 tvtower)
(connected room4 lakecross3)
(connected room4 lakepark)
(connected room4 mainavenue)
(connected room4 room3)
(connected room4 room5)
(connected room5 lakebuilding)
(connected room5 room3)
(connected room5 room4)
(connected room5 room6)
(connected room5 tvtower)
(connected room6 lakebuilding)
(connected room6 room1)
```

```
(connected room6 room5)
(connected sidelake lakecross1)
(connected sidelake lakepark)
(connected sidelake parking)
(connected sidelake policecross3)
(connected sidelake policecross4)
(connected sidelake policestation)
(connected sidelake sidelakestreet)
(connected sidelakestreet lakepark)
(connected sidelakestreet parking)
(connected sidelakestreet room2)
(connected sidelakestreet sidelake)
(connected tvtower room3)
(connected tvtower room5)
(named-point lakecross1-slot1)
(named-point lakecross1-slot2)
(named-point lakecross1-slot3)
(named-point lakecross1-slot4)
(named-point lakecross3-slot1)
(named-point lakecross3-slot2)
(named-point lakecross3-slot3)
(named-point lakecross3-slot4)
(named-point policecross1-slot1)
(named-point policecross1-slot2)
(named-point policecross1-slot3)
(named-point policecross1-slot4)
(named-point policecross1-slot5)
(named-point policecross1-slot6)
(named-point policecross2-slot1)
(named-point policecross2-slot2)
(named-point policecross2-slot3)
(named-point policecross3-slot1)
(named-point policecross3-slot2)
(named-point policecross3-slot3)
(named-point policecross3-slot4)
(named-point policecross4-slot1)
(named-point policecross4-slot2)
(named-point policecross4-slot3)
(named-point policecross4-slot4)
(place anywhere)
(place field)
(place lakebuilding)
(place lakecross1)
(place lakecross3)
(place lakepark)
(place mainavenue)
(place parking)
(place parkingbuilding)
(place policecross1)
(place policecross2)
(place policecross3)
(place policecross4)
(place policestation)
(place room1)
(place room2)
(place room3)
(place room4)
(place room5)
(place room6)
(place sidelake)
(place sidelakestreet)
(place tvtower)
(policeman self)
(protoagent a1)
(protoagent a10)
(protoagent a2)
(protoagent a3)
(protoagent a4)
(protoagent a5)
(protoagent a6)
(protoagent a7)
(protoagent a8)
(protoagent a9)
(strategic-point lakecross1 lakecross1-slot1)
(strategic-point lakecross1 lakecross1-slot2)
(strategic-point lakecross1 lakecross1-slot3)
(strategic-point lakecross1 lakecross1-slot4)
(strategic-point lakecross3 lakecross3-slot1)
(strategic-point lakecross3 lakecross3-slot2)
(strategic-point lakecross3 lakecross3-slot3)
(strategic-point lakecross3 lakecross3-slot4)
(strategic-point policecross1 policecross1-slot1)
(strategic-point policecross1 policecross1-slot2)
(strategic-point policecross1 policecross1-slot3)
(strategic-point policecross1 policecross1-slot4)
(strategic-point policecross1 policecross1-slot5)
(strategic-point policecross1 policecross1-slot6)
(strategic-point policecross2 policecross2-slot1)
(strategic-point policecross2 policecross2-slot2)
```

```
(strategic-point policecross2 policecross2-slot3)
(strategic-point policecross3 policecross3-slot1)
(strategic-point policecross3 policecross3-slot2)
(strategic-point policecross3 policecross3-slot3)
(strategic-point policecross3 policecross3-slot4)
(strategic-point policecross4 policecross4-slot1)
(strategic-point policecross4 policecross4-slot2)
(strategic-point policecross4 policecross4-slot3)
(strategic-point policecross4 policecross4-slot4)
)
(:goal
(and (at self lakecross1)
     (car-covered lakecross1-slot3)
     (car-covered lakecross1-slot1)
     (car-covered lakecross1-slot4)
     (car-covered lakecross1-slot2)))
)
```

C.3 Resulting plan

```
MOVE SELF ROOM2 PARKING
RECRUIT-HELP SELF A9
DELEGATED-DRIVE-AGENT A9 SELF ANYWHERE PARKING LAKECROSS1
DELEGATED-CAR-PROTECT-BUILDING-SLOT A9 LAKECROSS1-SLOT3
RECRUIT-HELP SELF A1
DELEGATED-MOVE A1 ANYWHERE LAKECROSS1
DECLARE-SLOT-CAR-PROTECTED SELF A9 LAKECROSS1-SLOT3
DELEGATED-CAR-PROTECT-BUILDING-SLOT A1 LAKECROSS1-SLOT1
RECRUIT-HELP SELF A10
DELEGATED-MOVE A10 ANYWHERE LAKECROSS1
DECLARE-SLOT-CAR-PROTECTED SELF A1 LAKECROSS1-SLOT1
DELEGATED-CAR-PROTECT-BUILDING-SLOT A10 LAKECROSS1-SLOT4
RECRUIT-HELP SELF A2
DELEGATED-MOVE A2 ANYWHERE LAKECROSS1
DELEGATED-CAR-PROTECT-BUILDING-SLOT A2 LAKECROSS1-SLOT2
DECLARE-SLOT-CAR-PROTECTED SELF A10 LAKECROSS1-SLOT4
DECLARE-SLOT-CAR-PROTECTED SELF A2 LAKECROSS1-SLOT2
```

C.4 Corresponding action in the virtual environment

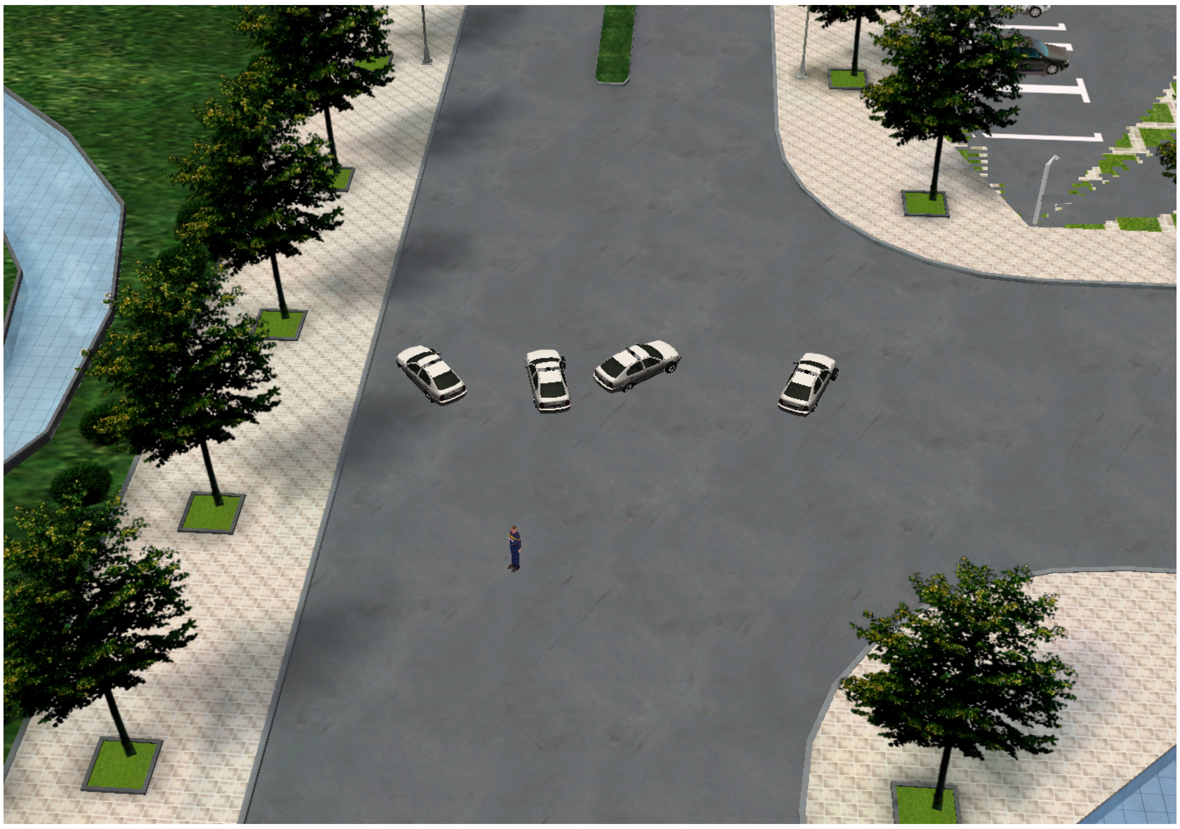


Figure C.1: Result of the plan execution in the virtual environment

Bibliography

[1] ABACI, T., CIGER, J., AND THALMANN, D. Action semantics in smart objects. In *Proceedings of the Workshop towards Semantic Virtual Environments* (Villars, Switzerland, March 2005), Miralab, pp. 120–162.

[2] ABACI, T., CIGER, J., AND THALMANN, D. Planning with smart objects. In *Proceedings of WSCG 05* (Pilsen, Czech Republic, 2005).

[3] ABACI, T., DE BONDELI, R., CÍGER, J., CLAVIEN, M., EROL, F., GUTIÉRREZ, M., NOVERRAZ, S., RENAULT, O., VEXO, F., AND THALMANN, D. Magic wand and the Enigma of the Sphinx. *Computers & Graphics 28* (2004), 477–484.

[4] ABACI, T., MORTARA, M., PATANÈ, G., SPAGNUOLO, M., VEXO, F., AND THALMANN, D. Bridging geometry and semantics for object manipulation and grasping. In *Proceedings of the Workshop towards Semantic Virtual Environment* (Villars sur Ollon, Switzerland, March 2005).

[5] ABELSON, H., SUSSMAN, J., AND SUSSMAN, J. *Structure and Interpretation of Computer Programs*, the second ed. MIT Press, 1984. ISBN: 0-262-01077-1.

[6] ALONSO, E., AND KUDENKO, D. Logic-based multi-agent systems for conflict simulations. In *Proceedings of UKMAS* (2000).

[7] ANDERSON, C. R., SMITH, D. E., AND WELD, D. S. Conditional effects in Graphplan. In *Proceedings of AIPS '98* (1998).

[8] BADLER, N., BINDIGANAVALE, R., BOURNE, J., PALMER, M., SHI, J., AND SCHULER, W. A parameterized action representation for virtual human agents. In *Embodied Conversational Agents* (Cambridge, MA, 2000), MIT Press, pp. 256–284.

[9] BAXTER, J., AND HEPPLEWHITE, R. A hierarchical distributed planning framework for simulated battlefield entities. In *Proceedings of the 19th Workshop of the UK Planning and Scheduling Special Interest Group (PLANSIG 2000)* (2000).

[10] BLUM, A. L., AND FURST, M. L. Fast planning through planning graph analysis. *Artificial Intelligence 90* (1997), 281–300.

[11] BLUMBERG, B. M. *Old Tricks, New Dogs: Ethology and Interactive Creatures*. PhD thesis, School Of Architecture And Planning, Massachusetts Institute of Technology, February 1997.

[12] BRATMAN, M. E. *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, 1987.

[13] CIGER, J., HERBELIN, B., AND THALMANN, D. Evaluation of gaze tracking technology for social interaction in virtual environments. In *2nd Workshop on Modelling and Motion Capture Techniques for Virtual Environments, CAPTECH04* (Zermatt, December 2004).

[14] COHEN, P. R., CHEYER, A. J., WANG, M., AND BAEG, S. C. An Open Agent Architecture. In *AAAI Spring Symposium* (March 1994), pp. 1–8. OAA.

[15] DARPA. Specification of the KQML agent communication language. Tech. rep., DARPA Knowledge Sharing Initiative, External Interfaces Working Group, 1993.

[16] DECKER, K., AND LESSER, V. Designing a family of coordination algorithms. Tech. rep., University of Massachusetts, Amherst, MA, USA, 1994.

[17] DECKER, K. S. *Environment Centered Analysis And Design of Coordination Mechanisms*. PhD thesis, Department of Computer Science, University of Massachusetts Amherst, May 1995.

[18] DORAN, J. E., FRANKLIN, S., JENNINGS, N. R., AND NORMAN, T. J. On cooperation in multi-agent systems. *Knowledge Engineering Review 12*, 3 (Sep 1997), 309–314.

[19] DOYLE, P., AND HAYES-ROTH, B. An intelligent guide for virtual environments. In *Proceedings of the 1st International Conference on Autonomous Agents* (New York, Feb. 5–8 1997), W. L. Johnson and B. Hayes-Roth, Eds., ACM Press, pp. 508–509.

[20] DURFEE, E. H. Organizations, plans, and schedules: An interdisciplinary perspective on coordinating AI agents. *Journal of Intelligent Systems*, Special Issue on the Social Context of Intelligent Systems (1991).

[21] DURFEE, E. H., AND LESSER, V. Partial global planning: A coordination framework for distributed hypothesis formation. *IEEE Transactions on Systems, Man, and Cybernetics 21*, 5 (September 1991), 1167–1183.

[22] EL-MANZALAWY, Y. Efficient planning with initial irrelevant facts. online, 2004. `http://www.cs.iastate.edu/~yasser/jplan.pdf`.

[23] EVERS, M., AND NIJHOLT, A. Jacob - an animated instruction agent in virtual reality. In *Advances in Multimodal Interfaces - ICMI 2000: Third International Conference. Lecture Notes in Computer Science*, T. Tan, Y. Shi, and W. Gao, Eds., vol. 1948. Springer-Verlag GmbH, 2000, ch. 526.

[24] FARENC, N., BOULIC, R., AND THALMANN, D. An informed environment dedicated to the simulation of virtual humans in urban context. In *Proceedings of Eurographics '99* (Milano, Italy, 1999), pp. 309–318.

[25] FIKES, R., AND NILSSON, N. J. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence 2* (1971), 189–208.

[26] FININ, T., FRITZSON, R., MCKAY, D., AND MCENTIRE, R. KQML as an agent communication language. In *CIKM '94: Proceedings of the third international conference on Information and knowledge management* (1994), ACM Press, pp. 456–463.

[27] FIPA. FIPA ACL Message Structure Specification. online, September 2005. `http://www.fipa.org/specs/fipa00061/`.

[28] FIPA. FIPA Communicative Act Library Specification. online, September 2005. `http://www.fipa.org/specs/fipa00037/`.

[29] FIPA. FIPA Contract Net Interaction Protocol Specification. online, February 2005. `http://www.fipa.org/specs/fipa00029/`.

[30] FIPA. FIPA Interaction Protocols. online, September 2005. `http://www.fipa.org/repository/ips.php3`.

[31] FIPA. FIPA KIF Content Language Specification. online, February 2005. `http://www.fipa.org/specs/fipa00010/XC00010B.html/`.

[32] FIPA. FIPA SL Content Language Specification. online, September 2005. `http://www.fipa.org/specs/fipa00008/`.

[33] FUNGE, J., TU, X., AND TERZOPOULOS, D. Cognitive modeling: Knowledge, reasoning and planning for intelligent characters. In *SIGGRAPH 99* (Los Angeles, CA, August 11-13 1999).

[34] GEIB, C., LEVISON, L., AND MOORE, M. B. SodaJack: An architecture for agents that search for and manipulate objects. Tech. Rep. MS-CIS-94-16LINC, University of Pennsylvania, LAB 265, Department of Computer and Information Science, University of Pennsylvania, December 1994.

[35] GIAMPAPA, J. A., AND SYCARA, K. Team-oriented agent coordination in the RETSINA multi-agent systems. In *1st International Joint Conference on Autonomous Agents and Multi-agent Systems (AAMAS 2002)* (Bologna, Italy, 2002), ACM Press.

[36] GRAHAM, J. R., DECKER, K. S., AND MERSIC, M. DECAF - a flexible multi agent system architecture. *Autonomous Agents and Multi-Agent Systems 7*, 1-2 (Jul 2003), 7.

[37] GRIFFITHS, N., AND LUCK, M. Coalition formation through motivation and trust. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems* (2003), ACM Press, pp. 17–24.

[38] GROSZ, B. J., AND KRAUS, S. The evolution of shared plans. In *Foundations of Rational Agency*, A. Rao and M. Wooldridge, Eds. Kluwer, Dordrecht, 1999, pp. 227–262.

[39] HELBING, D., AND MOLNÁR, P. Social force model for pedestrian dynamics. *Physical Review 51*, 5 (May 1995), 4282–4286 Part A.

[40] HILDEBRAND, M., NS, A. E., HUANG, Z., AND VISSER, C. Interactive agents learning their environment. In *Intelligent Virtual Agents: 4th International Workshop. Lecture Notes in Computer Science*, T. Rist, R. Aylett, D. Ballin, and J. Rickel, Eds., vol. 2792. Springer-Verlag GmbH, 2003, pp. 13–17.

[41] HOFFMANN, J. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research 20* (2003), 291–341.

[42] HOFFMANN, J., AND NEBEL, B. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research 14* (2001), 253–302.

[43] INGRAND, F. F., GEORGEFF, M. P., AND RAO, A. S. An architecture for real-time reasoning and system control. *IEEE Expert: Intelligent Systems and Their Applications 7*, 6 (1992), 34–44.

[44] IOERGER, T. R., AND JOHNSON, J. C. A formal model of responsibilities in agent-based teamwork. *Applied Artificial Intelligence 15*, 10 (November 2001), 875–916.

[45] JENNINGS, N. R. On being responsible. In *Decentralized AI 3*, E. Werner and Y. Demazeau, Eds. North-Holland, 1992, pp. 93–102.

[46] JENNINGS, N. R. Towards a cooperation knowledge level for collaborative problem solving. In *Proc. 10th European Conf. Artificial Intelligence, ECAI* (3–7 Aug. 1992), B. Neumann, Ed., John Wiley & Sons, pp. 224–228.

[47] JENNINGS, N. R. Commitments and conventions: The foundation of coordination in multi-agent systems. *Knowledge Engineering Review 8*, 3 (1993), 223–250.

[48] JENNINGS, N. R. Specification and implementation of a belief desire joint-intention architecture for collaborative problem solving. *Journal of Intelligent and Cooperative Information Systems 2*, 3 (1993), 289–318.

[49] JENNINGS, N. R. On agent-based software engineering. *Artificial Intelligence 117* (2000), 277–296.

[50] JUNG, H., AND TAMBE, M. *Conflicting agents: conflict management in multi-agent systems.* Kluwer Academic Publishers, 2001, ch. Conflicts in agent teams, pp. 153–167.

[51] KALLMANN, M. *Object Interaction in Real-Time Virtual Environments.* PhD thesis, École Polytechnique Fédérale de Lausanne, 2001.

[52] KAMINKA, G. A., AND TAMBE, M. Robust multi-agent teams via socially-attentive monitoring. *Journal of Artificial Intelligence Research 12* (2000), 105–147.

[53] KIM, I.-C. KGBot: A BDI agent deploying within a complex 3D virtual environment. In *Intelligent Virtual Agents. Lecture Notes In Artificial Intelligence*, T. Rist, R. Aylett, D. Ballin, and J. Rickel, Eds., vol. 2792. Springer-Verlag GmbH, 2003, pp. 192–196.

[54] KRAPICHLER, C., HAUBNER, M., ENGELBRECHT, R., AND ENGLMEIER, K.-H. VR interaction techniques for medical imaging applications. *Computer Methods and Programs in Biomedicine 56*, 1 (April 1998), 65–74.

[55] KRAUS, S., SHEHORY, O., AND TAASE, G. Coalition formation with uncertain heterogeneous information. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems* (New York, NY, USA, 2003), ACM Press, pp. 1–8.

[56] KRUM, D. M., OMOTESO, O., RIBARSKY, W., STARNER, T., AND HODGES, L. F. Speech and gesture multimodal control of a whole earth 3d visualization environment. In *VISSYM '02: Proceedings of the symposium on Data Visualisation 2002* (Aire-la-Ville, Switzerland, Switzerland, 2002), Eurographics Association, pp. 195–200.

[57] LABROU, Y., AND FININ, T. A proposal for a new KQML specification. Tech. Rep. TR-CS-97-03, Computer Science and Electrical Engineering Dept., Univ. of Maryland, Baltimore County, Baltimore, Md., 1997.

[58] LAIRD, J. E. It knows what you're going to do: Adding anticipation to a quakebot. In *Proceedings of the 5th Int. Conference on Autonomous Agents* (New York, 2001), ACM Press, pp. 385–392.

[59] LAIRD, J. E., NEWELL, A., AND ROSENBLOOM, P. S. Soar: an architecture for general intelligence. *Artificial Intelligence 33*, 1 (1987), 1–64.

[60] LANDER, S. E. *Distributed Search and Conflict Management Among Reusable Heterogeneus Agents*. PhD thesis, University of Massachusetts Amherst, 1994.

[61] LESPÉRANCE, Y., LEVESQUE, H., AND REITER, R. A situation calculus approach to modeling and programming agents. In *Foundations and Theories of Rational Agents*, A. Rao and M. Wooldridge, Eds. Kluwer, 1999.

[62] LEVISON, L. *Connecting planning and acting via object-specific reasoning*. PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, 1996.

[63] LEVISON, L., AND BADLER, N. How animated agents perform tasks: Connecting planning and manipulation through object-specific reasoning. In *Toward Physical Interaction and Manipulation* (1994), AAAI Spring Symposium Series.

[64] LUGER, G. F. *Artificial Intelligence*, 4. ed. Addison Wesley, 2002.

[65] MARSELLA, S., ADIBI, J., AL-ONAIZAN, Y., KAMINKA, G. A., MUSLEA, I., AND TAMBE, M. Experiences acquired in the design of robocup teams: A comparison of two fielded teams. *Autonomous Agents And Multi-Agent Systems 4*, 1-2 (Mar–Jun 2001), 115–129.

[66] MARSH, S., GHORBANI, A. A., AND BHAVSAR, V. C. The ACORN multi-agent system. *Web Intelli. and Agent Sys. 1*, 1 (2003), 65–86.

[67] MARTIN, C., SCHRECKENGHOST, D., BONASSO, P., KORTENKAMP, D., MILAM, T., AND THRONESBERY, C. An environment for distributed collaboration among humans and software agents. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems* (2003), ACM Press, pp. 1062–1063.

[68] MARTIN, D. L., CHEYER, A. J., AND MORAN, D. B. The Open Agent Architecture: A Framework for Building Distributed Software Systems. *Applied Artificial Intelligence 13*, 1/2 (1999), 91–128.

[69] MCDERMOTT, D. V. PDDL specification. online, 2004. version 2.1, `http://cs-www.cs.yale.edu/homes/dvm/`.

[70] MELO, F., CHOREN, R., CERQUEIRA, R., LUCENA, C., AND BLOIS, M. Deploying agents with the CORBA component model. In *Component Deployment:Second International Working Conference. Lecture Notes In Computer Science*, W. Emmerich and A. L. Wolf, Eds., vol. 3083. Springer-Verlag GmbH, 2004, pp. 234–247.

[71] MENEGUZZI, F. R., ZORZO, A. F., AND DA COSTA MÓRA, M. Propositional planning in BDI agents. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing* (New York, NY, USA, 2004), ACM Press, pp. 58–63.

[72] MILLER, M. S., YIN, J., VOLZ, R. A., IOERGER, T. R., AND YEN, J. Training teams with collaborative agents. *Lecture Notes In Computer Science 1839* (2000), 63–72.

[73] NAIR, R., TAMBE, M., AND MARSELLA, S. Role allocation and reallocation in multiagent teams: Towards a practical analysis. In *AAMAS 2003* (2003).

[74] NAIR, R., TAMBE, M., AND MARSELLA, S. Team formation for reformation in multiagent domains like RoboCupRescue. In *RoboCup 2002: Robot Soccer World Cup VI. Lecture Notes In Artificial Intelligence*, G. A. Kaminka, P. U. Lima, and R. Rojas, Eds., vol. 2752. Springer-Verlag GmbH, 2003, pp. 150–161.

[75] NEBEL, B., DIMOPOULOS, Y., AND KOEHLER, J. Irrelevant facts and operators in plan generation. Tech. Rep. 00089, Institut für Informatik Freiburg, April 1997.

[76] NIGENDA, R. S., NGUYEN, X., AND KAMBHAMPATI, S. Altalt: Combining the advantages of Graphplan and heuristic state search. In *Proceedings of KBCS-2000* (Mumbai, India, 2000).

[77] NIJHOLT, A., AND HULSTIJN, J. Multimodal interactions with agents in virtual worlds. In *Future Directions for Intelligent Information Systems and Information Science, Studies in Fuzziness and Soft Computing*, N. Kasabov, Ed. Physica-Verlag, 2000, pp. 148–173.

[78] PAYNE, T. R., SYCARA, K., AND LEWIS, M. Varying the user interaction within multi-agent systems. In *AGENTS '00: Proceedings of the fourth international conference on Autonomous agents* (2000), ACM Press, pp. 412–418.

[79] PENBERTHY, J. S., AND WELD, D. S. UCPOP: A sound, complete, partial-order planner for ADL. In *Third International Conference on Knowledge Representation and Reasoning (KR-92)* (Cambridge, MA, October 1992).

[80] PERLIN, K., AND GOLDBERG, A. Improv: a system for scripting interactive actors in virtual worlds. *Computer Graphics 30* (1996), 205–216.

[81] PERLIN, K., AND GOLDBERG, A. Improv: a system for scripting interactive actors in virtual worlds. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), ACM Press, pp. 205–216.

[82] PONDER, M., MOLET, T., PAPAGIANNAKIS, G., MAGNENAT-THALMANN, N., AND THALMANN, D. VHD++ development framework: Towards extendible, component based VR/AR simulation engine featuring advanced virtual character technologies. In *Computer Graphics International 2003* (2003), pp. 96–104.

[83] PYNADATH, D. V., KAMINKA, G. A., AND TAMBE, M. Monitoring teams by overhearing: A multi-agent plan-recognition approach. *Journal Of Artificial Intelligence Research 17* (2002), 83–135.

[84] PYNADATH, D. V., AND TAMBE, M. The communicative multiagent team decision problem: Analyzing teamwork theories and models. *Journal of Artificial Intelligence Research 16* (2002), 389–423.

[85] PYNADATH, D. V., TAMBE, M., ARENS, Y., CHALUPSKY, H., GIL, Y., KNOBLOCK, C., LEE, H., LERMAN, K., OH, J., RAMACHANDRAN, S., ROSENBLOOM, P. S., AND RUSS, T. Electric elves: Immersing an agent organization in a human organization. In *Proceedings of the AAAI Fall Symbolic on Socially Intelligent Agents* (2000).

[86] PYNADATH, D. V., TAMBE, M., CHAUVAT, N., AND CAVEDON, L. Toward team-oriented programming. In *Intelligent Agents*, N. R. Jennings and Y. Lespérance, Eds., vol. IV. Springer Verlag, 1999, pp. 233–247.

[87] RICH, C., AND SIDNER, C. L. COLLAGEN: When agents collaborate with people. In *Proceedings of the First International Conference on Autonomous Agents (Agents'97)* (New York, 5–8 1997), W. L. Johnson and B. Hayes-Roth, Eds., ACM Press, pp. 284–291.

[88] RICKEL, J., AND JOHNSON, W. L. Integrating pedagogical capabilities in a virtual environment agent. In *Proceedings of First International Conference on Autonomous Agents* (1997), ACM Press.

[89] RICKEL, J., AND JOHNSON, W. L. Virtual humans for team training in virtual reality. In *Artificial Intelligence in Education*, S. P. Lajoie and M. Vivet, Eds. IOS Press, Amsterdam, 1999, pp. 578–585.

[90] RICKEL, J., AND JOHNSON, W. L. Task-oriented collaboration with embodied agents in virtual worlds. In *Embodied Conversational Agents*, J. Cassell, J. Sullivan, S. Prevost, and E. Churchill, Eds. MIT Press, Cambridge, MA, 2000.

[91] RUSSELL, S. J., AND NORVIG, P. *Artificial intelligence: A Modern Approach*, 2nd ed. Prentice Hall, Englewood Cliffs, N.J., 2003.

[92] SCERRI, P., PYNADATH, D., JOHNSON, L., ROSENBLOOM, P., SI, M., SCHURR, N., AND TAMBE, M. A prototype infrastructure for distributed robot-agent-person teams. In *AAMAS 2003* (2003).

[93] SCHURR, N., OKAMOTO, S., MAHESWARAN, R. T., SCERRI, P., AND TAMBE, M. Evolution of a teamwork model. online, February 2005. `http://teamcore.usc.edu/schurr/papers/BDIBookChapter.pdf`.

[94] SCHWARTZ, D. G., STERLING, L. S., AND MAYLAND, E. The FLiPSiDE Blackboard: a financial logical programming system for distributed expertise. In *Proceedings of First International Conference on Artificial Intelligence on Wall Street* (9 Oct. 1991), IEEE, pp. 64–72. IEEE Catalog Number: 91TH0399-6.

[95] SEARLE, J. R. *Speech Acts*. Cambridge University Press, 1969.

[96] SERRANO, J. M., AND OSSOWSKI, S. On the impact of agent communication languages on the implementation of agent systems. In *Cooperative Information Agents VIII: 8th International Workshop. Lecture Notes in Computer Science*, M. Klusch, S. Ossowski, V. Kashyap, and R. Unland, Eds., vol. 3191. Springer-Verlag GmbH, January 2004, pp. 92–106.

[97] SERRANO, J. M., OSSOWSKI, S., AND FERNÁNDEZ, A. The pragmatics of software agents: Analysis and design of agent communication languages. In *Intelligent Information Agents: The AgentLink Perspective. Lecture Notes in Computer Science*, M. Klusch, S. Bergamaschi, P. Edwards, and P. Petta, Eds., vol. 2586. Springer-Verlag GmbH, January 2003, pp. 234–273.

[98] SHEHORY, O., AND KRAUS, S. Coalition formation among autonomous agents: Strategies and complexity. In *From Reaction to Cognition* (1995), no. 957, pp. 57–72.

[99] SYCARA, K., DECKER, K., PANNU, A., WILLIAMSON, M., AND ZENG, D. Distributed intelligent agents. *IEEE Expert, Intelligent Systems and their Applications 11*, 6 (1996), 36–45.

[100] TAMBE, M. Agent architectures for flexible, practical teamwork. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI-97/IAAI-97)* (Menlo Park, July 27–31 1997), AAAI Press, pp. 22–28.

[101] TAMBE, M. Towards flexible teamwork. *Artificial Intelligence 7* (1997), 83–124.

[102] TAMBE, M. Implementing agent teams in dynamic multiagent environments. *Journal of Applied Artificial Intelligence 12*, 2/3 (1998).

[103] TAMBE, M., ADIBI, J., ALONAIZON, Y., ERDEM, A., KAMINKA, G., MARSELLA, S., AND MUSLEA, I. Building agent teams using an explicit teamwork model and learning. *Artificial Intelligence 110*, 2 (1999).

[104] TAMBE, M., PYNADATH, D. V., CHAUVAT, N., DAS, A., AND KAMINKA, G. A. Adaptive agent integration architectures for heterogeneous team members. In *Proceedings of the ICMAS 2000* (2000), pp. 301–308.

[105] TAMBE, M., SHEN, W.-M., MATARIC, M., PYNADATH, D. V., GOLDBERG, D., MODI, P. J., QIU, Z., AND SALEMI, B. Teamwork in cyberspace: Using teamcore to make agents team-ready. In *Proceedings of the AAAI Spring Symposium on Agents in Cyberspace* (Menlo Park, CA, 1999), The AAAI Press.

[106] TAMBE, M., AND ZHANG, W. Towards flexible teamwork in persistent teams: Extended report. *Journal of Autonomous Agents and Multi-Agent Systems 3*, 2 (Jun 2000), 159–183.

[107] THIÉBAUX, S., HOFFMANN, J., AND NEBEL, B. In defense of PDDL axioms. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)* (Acapulco, Mexico, 2003), G. Gottlob, Ed.

[108] TU, X., AND TERZOPOULOS, D. Artificial fishes: Physics, locomotion, perception, behavior. In *Proceedings of SIGGRAPH 94* (New York, 1994), ACM Press, pp. 43–50.

[109] VELOSO, M., CARBONELL, J., PEREZ, A., BORRAJO, D., FINK, E., AND BLYTHE, J. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence 7*, 1 (1995).

[110] VOSINAKIS, S., AND PANAYIOTOPOULOS, T. A task definition language for virtual agents. *Journal of WSCG 11* (2003), 512–519.

[111] WELD, D. S., ANDERSON, C. R., AND SMITH, D. E. Extending Graphplan to handle uncertainty & sensing actions. In *Proceedings of AAAI '98* (1998).

[112] WOOLDRIDGE, M., AND JENNINGS, N. R. Formalizing the cooperative problem solving process. In *Proceedings of the Thirteenth International Workshop on Distributed Artificial Intelligence (IWDAI-94)* (Lake Quinalt, WA, July 1994), pp. 403–417.

[113] WOOLDRIDGE, M., AND JENNINGS, N. R. Towards a theory of cooperative problem solving. In *MAAMAW94* (Aug. 1994), pp. 15–26.

[114] YU, H., GHORBANI, A. A., BHAVSAR, V. C., AND MARSH, S. Keyphrase-Based Information Sharing in the ACORN Multi-Agent Architecture. In *Proceedings of the Second International Workshop on Mobile Agents for Telecommunication Applications (MATA 2000)* (Paris, France, 2000), E. Horlait, Ed., Springer-Verlag: Heidelberg, Germany, pp. 243–256.

Curriculum Vitae

General information

Name:	Ján Cíger
Date of birth:	27.9.1976, Slovak Republic
Nationality:	Slovak
Mother tongue:	Slovak
Other languages:	English, German, French, Russian, Czech

Education

Magister's Degree in Computer Science (Mgr.) with specialization in Computer Graphics and Parallel Algorithms and Distributed Systems. Faculty of Mathematics, Physics and Computer Science, Comenius University Bratislava, Slovak Republic, 1999

Professional activities

2001–2005

research assistant at VRlab (Virtual Reality Laboratory), EPFL, Switzerland

1999–2001

software engineer at WOC s.r.o, Slovak Republic

1998–1999

software engineer at BMS Group s.r.o, Slovak Republic

1996–1998

software engineer at UCS s.r.o, Slovak Republic

Publications

- J.Ciger, “Cheap and Accurate 3D Positioning Device for Virtual Reality Usage”, Proceedings of SCCG ’98 – Posters, p. 19–20, Budmerice, 1998
- J.Ciger, “An Ultrasonic Motion Tracker for VR Usage”, Proceedings of CESCG ’99, p. 163–170, Budmerice, 1999
- J.Ciger, “An Ultrasonic Motion Tracker for VR Usage”, Proceedings of SCCG ’99 – Posters, p. 41–42, Budmerice, 1999
- J.Ciger, “An Ultrasonic Motion Tracker for VR Usage”, in “CESCG ’97–’99 Selected Papers”, p. 175–182, Oesterreichische Computer Gesellschaft, Wien, 2000
- J.Ciger, J. Placek, “The Hand as an Ultimate Tool”, Proceedings of SCCG ’2000, p. 137–143, Budmerice, 2000
- J.Ciger, J. Placek, “Non-traditional image segmentation and filtering”, Proceedings of SCCG ’2001 – Posters, p. 25–27, Budmerice 2001
- J.Ciger, M. Gutierrez, F. Vexo, D.Thalmann, “The Magic Wand”, Proceedings of SCCG ’2003, p. 132–138, Budmerice, 2003
- T. Abaci, R. de Bondeli, J. Ciger, M. Clavien, F. Erol, M. Gutierrez, S. Noverraz, O. Renault, F. Vexo, D. Thalmann, “The Enigma of the Sphinx”, In International Conference on CYBERWORLDS, Singapore, 2003, pp. 106–113
- T. Abaci, R. de Bondeli, J. Ciger, M. Clavien, F. Erol, M. Gutierrez, S. Noverraz, O. Renault, F. Vexo, D. Thalmann, “Magic Wand and Enigma of the Sphinx”, Computers and Graphics, 2004
- T. Abaci, J.Ciger, D. Thalmann, “Speculative Planning With Delegation”, in International Conference on CYBERWORLDS, Tokyo, 2004
- Jan Ciger, Bruno Herbelin and Daniel Thalmann, “Evaluation of Gaze Tracking Technology for Social Interaction in Virtual Environments”, 2nd Workshop on Modelling and Motion Capture Techniques for Virtual Environments, CAPTECH04, Zermatt Dec. 9–11, 2004.
- Tolga Abaci, Jan Ciger, Daniel Thalmann, “Planning with Smart Objects”, Proceedings of WSCG ’05, Pilsen, 2005

- Tolga Abaci, Jan Ciger, Daniel Thalmann, “Action Semantics in Smart Objects”, in Workshop Towards Semantic Virtual Environments, SVE05, Villars, March 16–18, 2005